Symbolische Auswertung und Heuristiken zur Verifikation funktionaler Programme

Vom Fachbereich Informatik
der Technischen Universität Darmstadt
genehmigte

Dissertation

von Diplom-Informatiker Dirk Stephan Schweitzer
aus Frankfurt am Main

zur Erlangung des akademischen Grads eines
Doktor-Ingenieurs (Dr.-Ing.)

Erstreferent: Prof. Dr. Christoph Walther
Korreferent: Prof. Dr. Jürgen Giesl

Tag der Einreichung: 31.07.06
Tag der mündlichen Prüfung: 25.01.07

Hochschulkennziffer: D17

Darmstadt 2007

Danksagung

Ich möchte mich bei Prof. Dr. Christoph Walther für die Bereitstellung des Themas und die Betreuung bedanken. Außerdem danke ich Prof. Dr. Jürgen Giesl für seine Bereitschaft meine Dissertation als Korreferent zu begutachten.

Meinen Kollegen und Kolleginnen am Fachgebiet Programmiermethodik der TU Darmstadt Matthias Bormann, Carsten Kunz, Joachim Nadler, Markus Aderhold Andreas Schlosser und Veronika Weber danke ich für die freundschaftliche und tolle Arbeitsatmosphäre. Insbesondere Veronika Weber danke ich für ihre großartige Unterstützung in den letzten sieben Jahren bei der Erledigung der kleinen und großen Dingen im Fachgebietsalltag.

David Schahinian danke ich für das Korrekturlesen von Teilen der Arbeit. Für einen Geisteswissenschaftler war das Lesen der Arbeit sicher kein Vergnügen.

Mein Dank gilt auch meinem beiden „großen“ Brüdern Thomas und Alexander Schweitzer. Meinem Bruder Thomas danke für ein Angebot, dass ich zum Glück nicht annehmen musste. Bei meinem Bruder Alexander möchte ich mich für seinen fachlichen Rat und sein offenes Ohr bei der Ausarbeitung der Arbeit sowie das Korrekturlesen von Teilen der Arbeit bedanken.

Großen Dank schulde ich meinen Eltern Renate und Roland Schweitzer. Sie haben immer an mich geglaubt, mein Interesse an der Informatik gefördert und mir das Studium ermöglicht.

Schließlich möchte ich meiner Frau Sandra von ganzem Herzen danken. Ohne deine Liebe, dein Verständnis und deine Unterstützung wäre diese Arbeit nicht möglich gewesen. Du hast trotz aller Probleme nie das Vertrauen in mich verloren und gabst mir die Kraft um weiter zu machen. Du hast meine Launen und meine Unordnung ertragen, die Arbeit Korrektur gelesen, mir immer wieder den Rücken freigehalten und dabei auf vieles verzichtet. Für all dies und vieles mehr danke ich dir. Ich liebe dich.

Inhaltsverzeichnis

Kapitel 1

Einleitung

Informatikstudenten lernen Programmverifikation im Rahmen ihres Studiums in der Regel auf Basis von imperativen Programmen kennen. Hierbei müssen die Studenten meist Vor- und Nachbedingungen sowie Schleifeninvarianten für einfache Programme formulieren und dann mit Papier und Bleistift die Korrektheit der Programme auf Basis der formulierten Vor- und Nachbedingungen beweisen. Diese Beweise sind selbst für einfachste Programme zeitaufwendig und ausgesprochen fehleranfällig. Um komplexere Programme zu verifizieren ist daher der Einsatz entsprechender Verifikationswerkzeuge notwendig. Diese Werkzeuge befreien die Studenten von vielen der umfangreichen und lästigen Umformungen, die im Rahmen eines formalen Beweises auftreten und ermöglichen so, Zeit und Arbeit in die relevanten Beweisprobleme zu investieren. Der Einsatz mächtiger Systeme wie NQTHM [10], ACL2 [27], PVS [34], Isabelle [35], KIV [38], HOL [21], etc. im Rahmen einer Lehrveranstaltung ist jedoch aus folgenden Gründen problematisch:

- Die genannten Systeme basieren zum Teil auf komplexen Spezifikationssprachen. Dies führt dazu, dass die Studenten Zeit und Arbeit in das Erlernen der Spezifikationssprache investieren müssen und diese Zeit und Arbeit nicht in die Verifikation von Programmen investieren können.

- Die Bedienung der Systeme ist zum Teil gewöhnungsbedürftig. Die meisten der genannten Systeme sind im Wesentlichen „kommandozeilen-orientiert". Das bedeutet, dass der Benutzer entsprechende Beweiskommandos erlernen muss und diese dann über die Tastatur eingeben muss. Es ist daher für die Arbeit mit diesen Systemen notwendig, eine entsprechende Referenz der Beweiskommandos griffbereit zu haben, um die benötigten Beweiskommandos nachschauen zu können. Das bedeutet, dass das Erlernen der Bedienung des Systems entsprechend Zeit kostet.

- Die meisten Informatikstudenten besitzen einen Computer und möchten die Aufgabenstellungen zu Hause an ihrem eigenen Computer bearbeiten. Aus diesem Grund ist es notwendig, dass das eingesetzte Verifikationswerkzeug auf verschiedensten Computersystemen läuft und somit entsprechend portabel sein muss. Die meisten der genannten Systeme sind auf einige wenige Betriebssysteme festgelegt und dementsprechend wenig portabel.

Die Erfahrungen, die am Fachgebiet Programmiermethodik der Technischen Universität Darmstadt mit KIV in der Lehre gemacht wurden, bestätigen die genannten Probleme. Die Studenten kritisierten, dass ein erheblicher Teil der Lehrveranstaltung in das Erlernen der Spezifikationssprache und in die Bedienung des Verifikationswerkzeugs investiert werden musste. Diese Kritik der Studenten motivierte die Entwicklung des ✓eriFun-Systems. Das ✓eriFun-System wurde von Prof.

Dr. Christoph Walther in Zusammenarbeit mit dem Autor dieser Arbeit auf Basis der Diplomarbeit von Michael Schubart [42] am Fachgebiet Programmiermethodik der Technischen Universität Darmstadt entwickelt. Es handelt sich um einen in Java [3] implementierten semi-automatischen Induktionsbeweiser zur Verifikation funktionaler Programme, der unter besonderer Berücksichtigung der Anforderungen der Lehre entwickelt wurde. Das System weist insbesondere die folgenden Eigenschaften auf:

- **Einfache Logik**: Die dem ✓eriFun-System zugrunde liegende Logik ist recht einfach. Es handelt sich um eine Prädikatenlogik mit polymorphen Datentypen, die um ein Induktionsprinzip erweitert wurde. Da Studenten der Informatik üblicherweise Prädikatenlogik und polymorphe Datentypen im Grundstudium gelehrt wird, erfordert diese Logik eine sehr geringe Einarbeitungszeit für die Studenten. Sie können somit ohne lange Vorbereitungsphasen direkt mit dem System arbeiten.

- **Beweisautomatisierung**: Das System bietet ein hohes Maß an Beweisautomatisierung. Das bedeutet, dass viele der notwendigen Vereinfachungen und Umformungen, die während eines Beweises notwendig sind, durch das System automatisch durchgeführt werden. Diese Automatisierungen werden durch verschiedenste Heuristiken realisiert. Durch die Automatisierungen wird sichergestellt, dass die Studenten in der Regel lediglich bei nicht trivialen Beweisproblemen, deren Lösung eine gewisse Kreativität erfordert, durch entsprechende Interaktionen eingreifen müssen.

- **Benutzerfreundlichkeit**: Die graphische Oberfläche des Systems ist in Hinblick auf besondere Benutzerfreundlichkeit auf Basis der Programm-Bibliothek Swing [45] entwickelt worden. Alle Beweiskommandos werden durch entsprechende Menüpunkte ausgewählt. Die Parameter der Beweiskommandos müssen dann durch entsprechende Dialog-Fenster eingegeben werden, wobei die Eingabefelder bereits durch entsprechend sinnvolle Werte vorbelegt sind. Die Bedienung des ✓eriFun-Systems ist daher entsprechend einfach und intuitiv.

- **Portabilität**: Durch die Implementierung in Java ist das ✓eriFun-System für alle Computersysteme verfügbar, für die ein „*Java Runtime Environment*“ in der Version 5.0 existiert. Das System ist daher entsprechend portabel.

1.1 Ziel der Arbeit

Die vorliegende Arbeit beschreibt den im Rahmen des ✓eriFun-Systems entwickelten *symbolischen Auswertungskalkül* . Hierbei handelt es sich um einen Kalkül für Gleichheitsbeweise, welche typischerweise bei der Verifikation funktionaler Programme auftreten. Der Auswertungskalkül ist die vollautomatische Beweiskomponente des ✓eriFun-Systems und somit zu einem wesentlichen Teil für die Beweisautomatisierung verantwortlich.

Für Logikkalküle wie den Auswertungskalkül ist es üblich Eigenschaften wie Korrektheit, Vollständigkeit und (Nicht-)Entscheidbarkeit nachzuweisen. Weiter werden häufig Aussagen bezüglich der Komplexität behandelt. Solche Betrachtungen sind nicht Gegenstand dieser Arbeit.[1] Gegenstand der Arbeit sind vielmehr die Heuristi-

[1] Beim Auswertungskalkül handelt es sich um einen korrekten jedoch unvollständigen Kalkül. Korrektheit ist natürlich eine unabdingbare Forderung für jeden Logikkalkül. Auf einen Beweis der Korrektheit des Auswertungskalküls verzichtet wir jedoch, da die Auswertungsregeln von ihrem logischen Gehalt her im Wesentlichen durch Gleichheitsanwendungen definiert sind, deren Korrektheit leicht nachvollziehbar ist. Formale Beweise wären hier von ihrem qualitativen Gehalt Routine, vom quantitativen Gehalt jedoch so umfänglich, dass dies den Rahmen der Arbeit sprengen würde.

ken zur Anwendung der Auswertungsregeln, deren Repräsentation durch Kontrollinformationen sowie die Implementierung in einem Verifikationssystem. Ob die Heuristiken erfolgreich sind, lässt sich ausschließlich durch die Verifikation anspruchsvoller Fallstudien validieren. Bei der Entwicklung eines solchen Beweissystems muss ein pragmatischer Ansatz verfolgt werden. Das bedeutet, dass ausgehend von einem Prototypen Fallstudien bearbeitet werden. Während der Verifikation dieser Fallstudien, stößt der Prototyp auf Beweisprobleme, die er nicht lösen kann. Der Entwickler des Beweissystems muss dann die Ursachen des Scheiterns analysieren und den Prototypen entsprechend modifizieren. Diese Schritte (*Beweisen-Scheitern-Analysen-Modifizieren*) müssen mit zahlreichen sich bezüglich der Komplexität steigernden Fallstudien wiederholt werden, bis eine Stabilisierung der Definition des Beweissystems bzw. des zugrunde liegenden Kalküls eintritt. Der Entwicklungsprozess eines solchen Beweissystems kann nicht vollständig dargestellt werden. Vielmehr muss man sich darauf beschränken, die in der Analyse entdeckten Prinzipien anhand einfacher Beispiel zu illustrieren und so die Motivation der entwickelten Heuristiken darstellen. Genau dies ist Ziel und Inhalt dieser Arbeit. Insbesondere soll die Arbeit die Definition des Auswertungskalküls nachvollziehbar und transparent machen. Weiter gibt diese Arbeit eine vollständige formale Definition der im ✓eriFun-System verwendeten Programmiersprache und definiert so die formalen Grundlagen des Systems.

1.2 Aufbau der Arbeit

In Kapitel 2 geben wir einen kurzen Überblick über das ✓eriFun-System. Hierzu beschreiben wir informell die dem System zugrunde liegende Logik und stellen die vom System zur Verfügung gestellten Beweisregeln vor. Das Ziel dieses Kapitels ist es, die Funktions- und Arbeitsweise des ✓eriFun-Systems zu illustrieren und so das Verständnis für die nachfolgenden Kapitel zu erhöhen. Die restliche Arbeit gliedert sich dann in vier Teile.

Erster Teil Im ersten Teil beschreiben wir die formalen Grundlagen des ✓eriFun-System. Hierzu gehen wir wie folgt vor:

- In Kapitel 3 definieren wir die vollständige Syntax und Semantik der von ✓eriFun verwendeten Programmiersprache $\mathcal{L}$. Bei $\mathcal{L}$ handelt es sich um eine polymorph-getypte funktionale Programmiersprache ohne Funktionen höherer Ordnung. Die Programmiersprache ermöglicht außerdem die unvollständige Definition von Funktionen. Eine vollständige Beschreibung der Syntax und Semantik der Sprache ist notwendig, um den Auswertungskalkül formal definieren zu können.

- In Kapitel 4 definieren wir die von ✓eriFun verwendete Spezifikationssprache. Hierzu erweitern wir die in [51] und [52] definierte Spezifikationssprache auf das polymorphe Typsystem der Sprache $\mathcal{L}$. Dabei ist es unter anderem notwendig, die in [46] zur Repräsentation von Induktionsaxiomen eingeführten Relationenbeschreibungen auf das polymorphe Typsystem zu erweitern. Weiter geben wir auf Basis der Spezifikationssprache elementare Eigenschaften der durch ein Programm eingeführten Funktionssymbole an.

Zweiter Teil Im zweiten Teil der Arbeit definieren wir dann den Auswertungskalkül. Dieser Teil bildet den Hauptteil der Arbeit. Der Auswertungskalkül bildet – wie bereits erwähnt – die Grundlage für die vollautomatische Beweiskomponente des ✓eriFun-Systems und ist somit für die Beweisautomatisierung von entscheidender

Bedeutung. Der Auswertungskalkül formt auf Basis der Datenstrukturen und Funktionen eines Programms sowie unter Zuhilfenahme bereits verifizierter Aussagen das aktuelle Beweisproblem äquivalent um. Durch diese Umformung entsteht meist ein vereinfachtes Beweisproblem. Im Idealfall wird durch die Umformung das Beweisproblem sogar vollständig gelöst. Zur Definition des Auswertungskalküls gehen wir wie folgt vor:

- In Kapitel 5 beschreiben wir zunächst die Anforderungen, die der Auswertungskalkül als vollautomatische Beweiskomponente des ✓eriFun-Systems erfüllen muss. Weiter skizzieren wir, wie diese Anforderungen im Auswertungskalkül umgesetzt werden.
- In Kapitel 6 gehen wir auf die Kontrollinformationen ein, die der Auswertungskalkül benötigt, um einen Term effizient und zielgerichtet symbolisch auszuwerten. Hierbei unterscheiden wir Informationen, die die aktuelle Umgebung beschreiben unter der die Auswertung stattfinden soll (*Auswertungsumgebung*) und Informationen, die mit einem Term assoziiert sind, und so festlegen, wie die Informationen der Umgebung verwendet werden dürfen, um den Term auszuwerten (*Auswertungsannotationen*).
- In Kapitel 7 definieren wir dann den formalen Rahmen für den Auswertungskalkül. Die einzelnen Auswertungsregeln definieren wir in den nachfolgenden Kapiteln. In Kapitel 7 führen wir außerdem einige Konventionen und allgemeine Prinzipien bzgl. der Definition der Auswertungsregeln ein. Schließlich erläutern wir noch das weitere Vorgehen zur Definition der Auswertungsregeln.
- In den Kapiteln 8-13 führen wir dann die einzelnen Regeln des Auswertungskalküls ein. Die für eine erfolgreiche symbolische Auswertung zentralen Regeln sind hierbei die „*Execute*“- und „*Unfold*“-Regeln zur Auswertung von Prozeduraufrufen sowie die „*Assumption*“- und „*Replacement*“-Regeln zur Anwendung von Lemmata und Induktionshypothesen.

Dritter Teil Die Auswertungen des im zweiten Teil der Arbeit definierten Auswertungskalküls terminieren im Allgemeinen. Der Auswertungskalkül ist hierbei so definiert, dass die Ergebnisterme der Auswertungen in nachfolgenden Beweisschritten gut weiterverwendet werden können. Für einige seltene Fallkonstellationen sind jedoch Endlos-Auswertungen nicht ausgeschlossen. Auch für diese Fälle müssen wir aufgrund der Anforderungen aus Kapitel 5 die Terminierung gewährleisten. Wir definieren daher im dritten Teil der Arbeit ein Modifikation des Auswertungskalküls, die die Terminierung auch für diese Fälle gewährleistet und somit sicherstellt, dass der Auswertungskalkül allen Anforderungen aus Kapitel 5 genügt. Neben der Terminierung gehen wir im dritten Teil der Arbeit noch auf einen speziellen Aspekt der Verwendung des Auswertungskalküls - den so genannten Lemma-Filter - ein und diskutieren die Verwendung des ✓eriFun-Systems in Forschung und Lehre. Der dritte Teil gliedert sich somit insgesamt wie folgt:

- In Kapitel 14 modifizieren wir die Definition des Auswertungskalküls aus Kapitel 7, um die Terminierung des Auswertungskalküls zu gewährleisten. Hierzu beschränken wir die maximale Anzahl der Schritte einer symbolischen Auswertung. Wichtig ist hierbei, dass keine Modifikationen der Auswertungsregeln aus den Kapiteln 8-13 notwendig sind, um die Terminierung sicherzustellen.
- In Kapitel 15 definieren wir den Lemma-Filter. Dieser Lemma-Filter ist für die Effizienz des symbolischen Auswertungskalküls und somit für die Effizienz des Systems insgesamt von entscheidender Bedeutung. Der Lemma-Filter

bestimmt, welche der vom ✓eriFun-System bereits verifizierten Aussagen zur Vereinfachung des aktuellen Beweisproblems vom Auswertungskalkül verwendet werden dürfen. Durch den Lemma-Filter wird somit der Suchraum eingeschränkt. Die Herausforderung bei Definition dieser Heuristik ist, den Suchraum möglichst stark einzuschränken ohne jedoch erforderliche Lemmata zu verlieren.

- In Kapitel 16 beschreiben wir den Einsatz des ✓eriFun-Systems in Forschung und Lehre. Hierbei spielt der Einsatz in der Lehre eine Hauptrolle, da das System unter spezieller Berücksichtigung der Lehre entwickelt wurde. Wir geben daher einen kurzen Bericht über die in der Lehre gesammelten Erfahrungen. Insbesondere geben wir eine Übersicht über die bisher mit dem System gerechneten Fallstudien. Weiter beschreiben wir in Kapitel 16 die sich aktuell in Entwicklung befindlichen Erweiterungen des Systems. Schließlich geben wir noch einige Ideen für zukünftige Erweiterungen des Systems im Allgemeinen als auch des Auswertungskalküls im Speziellen.

Vierter Teil Der vierte Teil besteht aus dem Anhang. Er gliedert sich wie folgt.

- In Anhang A definieren wir einige der von uns verwendeten mathematischen Grundlagen bzgl. Mengen, Listen, Relationen und Funktionen. Wir setzen im weiteren Verlauf dieser Arbeit die Begriffe und Notationen dieses Anhangs voraus.
- In Anhang B beschreiben wir, wie mit Hilfe des Auswertungskalküls auf Basis von Relationenbeschreibungen die Induktionsbasis- und Induktionsschrittfälle eines Induktionsbeweises berechnet werden.
- In Anhang C wird die Funktion $\mathit{perm\text{-}flat}_{\mathcal{U},f}$ definiert, auf deren Basis die so genannten „*Functionality*“-Regeln des Auswertungskalküls definiert sind.
- In Anhang E definieren wir Auswertungsregeln, die zwar in der Implementierung des Auswertungskalküls durch das ✓eriFun-System enthalten sind, deren Nützlichkeit aber durch unsere Fallstudien nicht mehr belegt wird. Das heißt, dass die Definition der Regeln für die Fallstudien nicht mehr erforderlich ist. Diese Regeln sind daher nicht Teil unserer Definition des Auswertungskalküls.
- In Anhang F haben wir Beispiele zusammengefasst, die aus Gründen der Übersichtlichkeit nicht den Kapiteln 8-13 enthalten sind. Es sind insbesondere Beispiele für Überauswertungen und Endlos-Auswertungen.

1.3 Anmerkungen

Es ist zu beachten, dass alle aktuellen Beschreibungen des ✓eriFun-Systems wie beispielsweise [50, 51, 52, 56, 59] sich auf die Versionen 2.5.5, 2.5.6 bzw. 2.6.1 des ✓eriFun-Systems beziehen. Die Version 3.0, auf der diese Arbeit basiert, erweitert jedoch die verwendete Programmiersprache und damit auch die Spezifikationssprache um eine Reihe von Eigenschaften:

- Polymorphe Datenstrukturen und Funktionen.
- Unvollständig-definierte Funktionen.
- Strukturelle Fallunterscheidungen.
- Definition lokaler Variablen durch `let`-Ausdrücke.

Sowohl die Programmiersprache also auch die Spezifikationssprache der Version 3.0 des Systems sind somit in keiner bisherigen Veröffentlichung vollständig beschrieben. Insbesondere wird das polymorphe Typsystem in Kombination mit unvollständig-definierten Funktionen erst in dieser Arbeit formal spezifiziert. Die Kapitel 3 und 4 schließen somit eine Lücke in der aktuellen Dokumentation des ✓eriFun-Systems und dienen für sich genommen als Sprachreferenz der Version 3.0 des ✓eriFun-Systems.

Schließlich sei darauf hingewiesen, dass bereits eine teilweise Beschreibung des Auswertungskalküls existiert [50]. Diese Beschreibung ist jedoch unvollständig. So werden zum einen nicht alle Regeln des Auswertungskalküls beschrieben und zum anderen die verwendeten Heuristiken und Kontrollmechanismen des Kalküls teilweise nur unzureichend bzw. unvollständig dargestellt. Weiter ist zu bemerken, dass sich [50] auf die Version 2.6.1 des ✓eriFun-Systems bezieht. Dementsprechend sind die Konsequenzen, die sich aufgrund der neuen Spracheigenschaften der Version 3.0 für den Auswertungskalkül ergeben, dort nicht berücksichtigt. Schließlich wurden einige Regeln des Kalküls während der Entwicklung von Version 2.6.1 zur Version 3.0 stark modifiziert, so dass die Beschreibungen der entsprechenden Regeln in [50] nicht mehr aktuell sind. Nichtsdestotrotz baut die vorliegende Arbeit in mancher Hinsicht auf [50] auf. Insbesondere wurden einige der Beispiel aus [50] übernommen, wobei diese teilweise modifiziert wurden.

Kapitel 2

Das VeriFun-System

Dieses Kapitel gibt eine Einführung in die Funktionsweise und das Arbeiten mit dem ✓`eriFun`-System. Es ist im Wesentlichen eine Zusammenfassung und Aktualisierung des dritten Kapitels von [58]. Für eine ausführliche Darstellung des Systems verweisen wir daher insbesondere auf diese Veröffentlichung sowie auf [51, 52, 56].[1]

Das Kapitel gliedert sich wie folgt. In den Abschnitten 2.1 und 2.2 beschreiben wir kurz die dem System zugrunde liegende Logik. Hierzu erklären wir, wie mit dem System Datenstrukturen und Prozeduren definiert, Aussagen über diese Datenstrukturen und Prozeduren formuliert und schließlich verifiziert werden können. In Abschnitt 2.3 beschreiben wir, wie sich das System dem Benutzer präsentiert. Anschließend erklären wir in Abschnitt 2.4 die vom ✓`eriFun`-System zur Verfügung gestellten Beweisregeln und beschreiben dann in Abschnitt 2.5 das Zusammenspiel von Interaktionen des Benutzers und Automatisierungen durch das System. Schließlich illustrieren wir in Abschnitt 2.6 anhand eines vollständigen Beispiels wie das ✓`eriFun`-System mit Hilfe des Auswertungskalküls die Gültigkeit eines Lemmas zeigt.

2.1 Annotierte funktionale Programme

Programme werden in ✓`eriFun` in der Programmiersprache $\mathcal{L}$ formuliert. Bei $\mathcal{L}$ handelt es sich um eine einfache, polymorph getypte, funktionale Programmiersprache ohne Prozeduren höherer Ordnung. Programme der Sprache $\mathcal{L}$ bestehen aus *Typoperator-* und *Prozedurdefinitionen*. Bei Typoperatordefinitionen handelt es sich um ein einfaches Definitionsprinzip auf Basis von Konstruktoren und Selektoren zur Definition freier algebraischer Datenstrukturen. Die durch Typoperatordefinitionen eingeführten Datenstrukturen werden als *Typoperatoren* bezeichnet. Beispielsweise definiert die folgende Typoperatordefinition einen Typoperator `list` zur Repräsentation endlicher Listen:[2]

$$\begin{array}{l}\texttt{structure list[@a]} \Leftarrow \\ \quad \texttt{empty, add(hd : @a, tl : list[@a])}.\end{array} \tag{2.1}$$

Mit Hilfe dieser Definition können endliche Listen durch die Konstruktoren `empty` und `add` erzeugt werden. Auf die Komponenten einer Liste, d.h. auf den Kopf der Liste und die Restliste, kann dann mit Hilfe der Selektoren `hd` und `tl` zugegriffen werden. Für jeden Konstruktor *cons* wird durch eine Typoperatordefinition zusätzlich ein *Strukturprädikat* `?`*cons* definiert. Mit Hilfe dieser Strukturprädikate kann

[1] Die Veröffentlichungen beziehen sich die Versionen 2.5.5, 2.5.6 bzw. 2.6.1.

[2] Mit `@a, @b, ...` werden in ✓`eriFun` die Typvariablen polymorpher Typen bezeichnet.

dann überprüft werden, ob es sich bei einem Term um einen *cons*-Wert handelt oder nicht. Für die Typoperatordefinition (2.1) werden beispielsweise die Strukturprädikate `?empty` und `?add` erzeugt. Der Term `?add(empty)` wird dann zu `false` ausgewertet und der Term `?add(add(n,l))` zu `true`.

Die Typoperatoren `bool` und `nat` sind durch die Sprache $\mathcal{L}$ vordefiniert:

$$\texttt{structure bool} \Leftarrow \texttt{true,false},$$
$$\texttt{structure nat} \Leftarrow \texttt{0,succ(pred : nat)}.$$

Diese Typoperatoren repräsentieren die boolschen Wahrheitswerte und die natürlichen Zahlen. Die boolschen Wahrheitswerte werden durch die Konstruktoren `true` und `false` dargestellt. Die natürlichen Zahlen werden mit Hilfe der Konstruktoren `0` und `succ` erzeugt. So repräsentiert beispielsweise der Term `succ(succ(0))` die natürliche Zahl 2.

Prozeduren über den Typoperatoren werden durch die Prozedurdefinitionen formuliert. Diese Prozedurdefinitionen bestehen aus einem *Prozedurkopf* und einem *Prozedurrumpf*. Der Prozedurkopf legt den Namen, die formalen Parameter, deren Typen sowie den Rückgabetyp der Prozedur fest. Der Prozedurrumpf definiert den eigentlichen Algorithmus. Er wird in Form eines „*first-order*“ Terms basierend auf Fallunterscheidungen, Funktionsapplikationen, `let`-Ausdrücken und Rekursion angegeben. Ein Beispiel für eine Prozedurdefinition ist die folgende Definition von `>` zum Vergleich natürlicher Zahlen:[3]

```
function >(x, y : nat) : bool ⇐
  if(x=0,
     false,
     if(y=0,
        true,
        pred(x)>pred(y))).
```

Aussagen über Programme werden in ✓`eriFun` in Form von *Lemmata* angeben:[4]

$$\texttt{lemma}\ \mathit{name} \Leftarrow \texttt{all}\ x_1 : \tau_1, \ldots, x_n : \tau_n\ \mathit{body}. \tag{2.2}$$

Die x_i bezeichnen hierbei Variablen und die τ_i die Typen dieser Variablen. *body* bezeichnet einen boolschen Term, der die eigentliche Aussage repräsentiert. Zur Formulierung von Junktoren wird das Konditional `if` verwendet. Weiter müssen alle im Term *body* enthaltenen Termvariablen durch den Quantor `all` gebunden werden. Das bedeutet, dass mit Lemmata ausschließlich all-quantifizierte Aussagen formuliert werden können. Die Transitivität der Prozedur `>` lässt sich beispielsweise durch folgendes Lemma ausdrücken:

```
lemma >_transitiv ⇐ all x, y, z : nat
  if(x>y, if(y>z, x>z, true), true)
```
(2.3)

Eine Liste von Typoperatordefinitionen, Prozedurdefinitionen und Lemmata wird als *annotiertes Programm* bezeichnet. Das eigentliche Programm ist hierbei durch die Lemmata annotiert. Ein Beispiel für ein annotiertes Programm ist in Abbildung 2.1 dargestellt. Die Prozedur `minsort` sortiert dabei auf Basis des Minsort-Sortierverfahrens eine Liste natürlicher Zahlen. Die Prozedur `minimum` ermittelt das Minimum der übergebenen Liste `l`. Es ist zu beachten, dass das Minimum einer leeren Liste nicht definiert ist. Die Prozedur `minimum` lässt daher den

[3] Das Funktionssymbol `>` verwenden wir in Infix-Schreibweise.

[4] Wir verwenden in diesem Kapitel die im ✓`eriFun`-System tatsächlich verwendete Syntax von Lemmata. Die in der restlichen Arbeit verwendete Syntax von Lemmata, führen wir in Kapitel 4 ein. Diese Syntax quantifiziert auch die im Lemma verwendeten Typvariablen.

```
structure list[@a] ⇐
  empty, add(hd : @a, tl : list[@a])

function delete(n : @a, l : list[@a]) : list[@a] ⇐
  if(?empty(l),
     empty,
     if(n=hd(l),
        delete(n, tl(k)),
        add(n, delete(n, tl(k)))))

function minimum(l : list[nat]) : nat ⇐
  if(?empty(l),
     *,
     if(?empty(tl(l)),
        hd(l),
        if(hd(l)>minimum(tl(l)),
           minimum(tl(l)),
           hd(l))))

function ordered(l : list[nat]) : bool ⇐
  if(?empty(l),
     true,
     if(?empty(tl(l)),
        true,
        if(hd(l)>hd(tl(l)),
           false,
           ordered(tl(k)))))

function occurs(n : nat, k : list[nat]) : nat ⇐
  if(?empty(k),
     0,
     if(n=hd(k),
        succ(occurs(n, tl(k))),
        occurs(n, tl(k))))

function minsort(l : list[nat])list[nat] ⇐
  if(?empty(l),
     empty,
     let m:=minimum(l) in
       add(m, minsort(delete(m, l)))
     end)

lemma minsort_sorts ⇐ all l : list[nat]
  ordered(minsort(l))

lemma minsort_permutes ⇐ all n : nat, l : list[nat]
  occurs(n, l)=occurs(n, minsort(l))
```

Abbildung 2.1: Beispiel eines annotierten Programms bestehend aus Datenstrukturen, Prozeduren und Lemmata.

```
structure tree[@a] ⇐
  nil,
  leaf(value : @a),
  node(left : tree, right : tree)

function elem(v : @a, t : tree[@a]) : bool ⇐
  case(t,
       nil : false,
       leaf : value(t) = v,
       node : if(elem(v, left(t)), true, elem(v, right(t))))

lemma nil_is_empty ⇐ all v : @a
  if(elem(v, nil), false, true)
```

Abbildung 2.2: Beispiel eines annotierten Programms mit `case`.

Rückgabewert für den Fall, dass die leere Liste übergeben wird, offen. Dies wird durch den Rückgabewert $*$ ausgedrückt. Bei `minimum` handelt es sich daher um eine unvollständig-definierte Prozedur.

Das es sich bei `minsort` tatsächlich um ein Sortierverfahren handelt, wird durch die Lemmata `minsort_sorts` und `minsort_permutes` ausgedrückt. Aus der Gültigkeit des Lemmas `minsort_sorts` folgt, dass das Ergebnis von `minsort` eine sortierte Liste ist. Aus der Gültigkeit des Lemmas `minsort_permutes` folgt, dass das Ergebnis von `minsort` eine Permutation der übergebenen Liste ist. Um diese beiden Lemmata zu formulieren, ist es notwendig die zusätzlichen Prozeduren `occurs` und `ordered` zu definieren. `occurs` ermittelt hierbei die Anzahl des Elements `n` in der Liste `l` und `ordered` überprüft, ob die übergebene Liste sortiert ist.

Ein weiteres Beispiel für ein annotiertes Programm ist in Abbildung 2.2 dargestellt. Die Prozedur `elem` überprüft, ob `v` im übergebenen Baum `t` enthalten ist. Im Gegensatz zu den Prozeduren in Abbildung 2.1 ist die Fallunterscheidung in `elem` mit Hilfe eines `case` definiert. `case` führt eine Fallunterscheidung bzgl. der Konstruktoren der verwendeten Datenstruktur durch. Für die Fallunterscheidung der Prozedur `elem` ist dies eine Fallunterscheidung bzgl. der Konstruktoren `nil`, `leaf` und `node`.

2.2 Beweise von Programmaussagen

Die Gültigkeit von Lemmata wird in ✓`eriFun` üblicherweise mit Hilfe von Induktion bewiesen. Die zum Beweis eines Lemmas erzeugten Beweisverpflichtungen repräsentieren daher im Allgemeinen die Basis- und Schrittfälle einer Induktion. ✓`eriFun` stellt die Beweisverpflichtungen durch so genannte *Sequenzen* dar:

$$H, IH \vdash g$$

H bezeichnet hierbei eine Menge von Literalen, die die durch die Induktion definierte Fallunterscheidung repräsentieren. Ein *Literal* ist ein Atom oder ein negiertes Atom. Als *Atom* bezeichnen wir einen `if`-freien boolschen Term $\neq$ `false`. Für ein negiertes Atom `if`$(a, \texttt{false}, \texttt{true})$ schreiben wir kurz $\neg a$. IH bezeichnet die Menge der für die Sequenz gültigen Induktionshypothesen und g den so genannten Goal-Term der Sequenz, der die Induktionskonklusion repräsentiert. Induktionshypothesen und Goal-Terme sind boolsche Terme. Um beispielsweise die Gültigkeit des Lemma (2.3) zu beweisen, ist eine strukturelle Induktion über die Variable `x`

erfolgreich. ✓`eriFun` repräsentiert den Induktionsanfang und den Induktionsschritt dieses Induktionsbeweises durch die folgenden Sequenzen:

$$\{\texttt{?0(x)}\}, \emptyset \vdash \texttt{if(x>y, if(y>z, x>z, true), true)},$$

$$\{\neg\texttt{?0(x)}\}, \{\texttt{all y,z : nat if(pred(x)>y, if(y>z, pred(x)>z, true), true)}\} \vdash \texttt{if(x>y, if(y>z, x>z, true), true)}.$$

Die Menge aller Sequenzen definiert die Sprache des so genannten HPL-Kalküls.[5] Mittels des HPL-Kalküls wird die Gültigkeit von Lemmata in ✓`eriFun` bewiesen. Die Anwendung einer Beweisregel auf eine Sequenz *seq* erzeugt eine endliche Menge von Sequenzen, die durch Modifikationen der Sequenz *seq* entstehen. Diese Sequenzen werden als *Kindsequenzen* der Sequenz *seq* bezeichnet und die Sequenz *seq* selbst als *Muttersequenz*. Die Gültigkeit der Kindsequenzen impliziert dann die Gültigkeit der Muttersequenz. Zum Beweis eines Lemmas der Form (2.2) erzeugt das System zunächst eine Sequenz der Form

$$\emptyset, \emptyset \vdash \mathit{body}\,.$$

Durch Anwenden von Beweisregeln des HPL-Kalküls auf diese Sequenz bzw. auf die Kinder bzw. Kindeskinder der Sequenz entsteht dann ein Beweisbaum. Ein Beweisbaum wird als *geschlossen* bezeichnet, falls alle Blätter des Beweisbaums durch Sequenzen mit Goal-Term `true` gegeben sind. Ein geschlossener Beweisbaum ist *abgeschlossen*, falls alle zur Konstruktion des Beweisbaums verwendeten Lemmata durch das ✓`eriFun`-System verifiziert sind. Ein Lemma heißt *verifiziert*, falls der mit dem Lemma assoziierte Beweisbaum abgeschlossen ist. Abgeschlossene Beweisbäume und verifizierte Lemmata sind somit offensichtlich gegenseitig rekursiv-definiert. Den Rekursionsanfang bilden dabei die Beweisbäume, die keine Lemmata verwenden.

Der HPL-Kalkül definiert insgesamt 14 Beweisregeln zur Konstruktion von Beweisbäumen. Die Beweisregel „*Induction*“ erzeugt beispielsweise für eine Sequenz entsprechende Basis- und Schrittfälle, und die Beweisregel „*Use Lemma*“ wendet ein Lemma auf eine Sequenz an. Die Goal-Terme von Sequenzen werden durch Anwendung der so genannten „*Computed Proof Rules*“ vereinfacht. Hierbei werden die Goal-Terme mit Hilfe des *symbolischen Auswertungskalküls* ausgewertet. Wir sprechen dann von einer symbolischen Auswertung des Goal-Terms. Der Auswertungskalkül verwendet hierzu die Typoperator- und Prozedurdefinitionen des aktuellen Programms, die bereits verifizierten Lemmata sowie die Hypothesen und Induktionshypothesen der Sequenz des Goal-Terms. Um eine effiziente und zielgerichtete symbolische Auswertung zu ermöglichen, sind die Goal-Terme der Sequenzen mit Zusatzinformationen, so genannten *Annotationen* versehen. Diese Annotationen steuern die symbolische Auswertung des Goal-Terms. So sind beispielsweise Prozeduraufrufe mit einem so genannten Label markiert. Dieses Label bestimmt, ob der Prozeduraufruf ausgewertet werden darf oder nicht. Eine andere Annotation ist das so genannte Search-Limit. Dieses Search-Limit spezifiziert die Größe des Suchraums zur Anwendung von Lemmata. Ist beispielsweise das Search-Limit auf 0 gesetzt, so wird der betreffende Term während der symbolischen Auswertung nicht mit Hilfe von Lemmata vereinfacht.

Das ✓`eriFun`-System ermöglicht ausschließlich die Verifikation von Lemmata über terminierenden Prozeduren. Das bedeutet, dass die Prozeduren eines Lemmas vor der Verifikation des Lemmas als terminierend nachgewiesen werden müssen. Hierzu generiert das System für jede Prozedur eine Menge von Terminierungshypothesen, die die Terminierung der Prozedur gewährleisten. Diese Terminierungshypothesen werden dann wie Lemmata mit Hilfe des HPL-Kalküls verifiziert.

[5] HPL ist eine Abkürzung für „*Hypotheses, Programs and Lemmas*“.

2.3 Beweisen in VeriFun

Das ✓`eriFun`-System präsentiert sich nach einem Start durch das so genannte Hauptfenster (Abbildung 2.3). Dieses Hauptfenster ist in zwei Teile unterteilt. Im linken Teil des Hauptfensters wird das aktuelle annotierte Programm dargestellt. Das annotierte Programm kann dabei durch Verzeichnisse strukturiert werden. So enthält beispielsweise jedes annotierte Programm ein Verzeichnis „*Predefined*“, welches alle vom ✓`eriFun`-System vordefinierten Programmelemente enthält. Dies sind die Typoperatordefinitionen für `bool` und `nat`, die Prozedurdefinition für `>` und eine Reihe von Lemmata über `nat` und `>`. Der Benutzer kann mit Hilfe des Menüpunkts **Program\Insert** neue Programmelemente in das aktuelle annotierte Programm einfügen.

Der rechte Teil des Hauptfensters stellt den Beweisbaum des aktuell ausgewählten Programmelements dar. Da mit Typoperatordefinitionen keine Beweisbäume assoziiert sind, wird bei der Auswahl einer Typoperatordefinition auch kein Beweisbaum im rechten Teil des Hauptfensters dargestellt. Für Prozedurdefinitionen wird der Beweisbaum einer Terminierungshypothese dargestellt.

Den Knoten der Beweisbäume, d.h. den Sequenzen, wird durch das System ein *Beweisstatus* zugewiesen. Dieser Beweisstatus – auch „*Proof State*“ genannt – gibt Auskunft über den aktuellen Entwicklungszustand des durch den Knoten definierten Teilbeweisbaums. Der Beweisstatus eines Knotens wird durch eine farbige Markierung dargestellt und automatisch aktualisiert. Die möglichen Zustände sind:

- „*proved*“: Ist der Teilbaum des Knotens ein abgeschlossener Beweisbaum, so erhält der Knoten den Beweisstatus „*proved*“. Dieser Beweisstatus wird durch eine grüne Markierung dargestellt.

- „*completed*“: Ist der Teilbaum des Knotens ein geschlossener, jedoch nicht abgeschlossener Beweisbaum, so erhält der Knoten den Beweisstatus „*completed*“. Dieser Beweisstatus wird durch eine gelbe Markierung dargestellt.

- „*unprovable*“: Kann der Teilbaum eines Knotens nicht mehr zu einem geschlossenen Beweisbaum entwickelt werden, so wird dem Knoten durch das System der Beweisstatus „*unprovable*“ zugewiesen. Es ist dann notwendig, diesen Teil des Beweisbaums mit Hilfe des Menüpunkts **Proof\Prune** abzuschneiden und einen neuen Teilbaum zu konstruieren. Der Beweis-Zustand „*unprovable*“ wird durch eine violette Markierung dargestellt.

- „*disproved*“: Wird durch den Beweisbaum die Ungültigkeit der Sequenz bewiesen, so wird dem Knoten durch das System der Beweisstatus „*disproved*“ zugewiesen. Dieser Beweisstatus wird mit einer roten Markierung dargestellt.

- „*unproved*“: Kann dem Knoten keiner der anderen Beweis-Zustände zugewiesen werden, so wird der Beweisstatus „*unproved*“ zugewiesen. Dieser Beweisstatus wird mit einer blauen Markierung dargestellt.

Den Programmelementen eines annotierten Programms wird durch ✓`eriFun` ein *Programmstatus* zugewiesen. Dieser Programmstatus – auch „*Program State*“ genannt – gibt Auskunft über den aktuellen Verifikationszustand des Programmelements. Der Programmstatus wird durch eine farbige Markierung des Programmelements dargestellt, die automatisch aktualisiert wird. Ein Lemma hat den Programmstatus „*ignored*“, falls die Terminierung einer der aufgerufenen Prozeduren noch nicht bewiesen ist. Ansonsten wird der Programmstatus des Lemmas mit Hilfe der folgenden Tabelle aus dem Beweisstatus des mit dem Lemma assoziierten Beweisbaums abgeleitet:

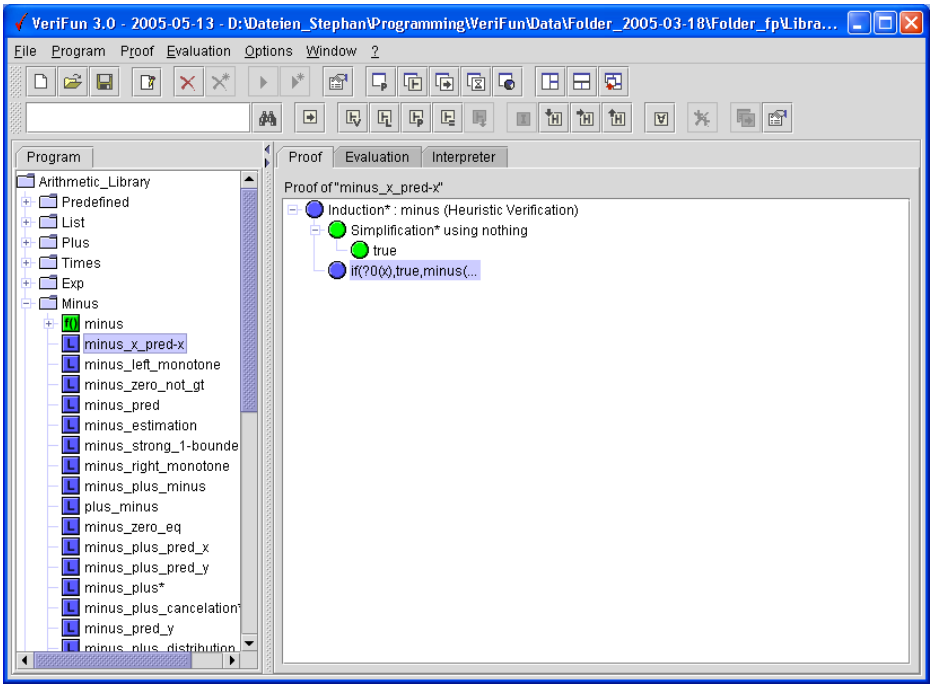


Abbildung 2.3: Das Hauptfenster von ✓eriFun.

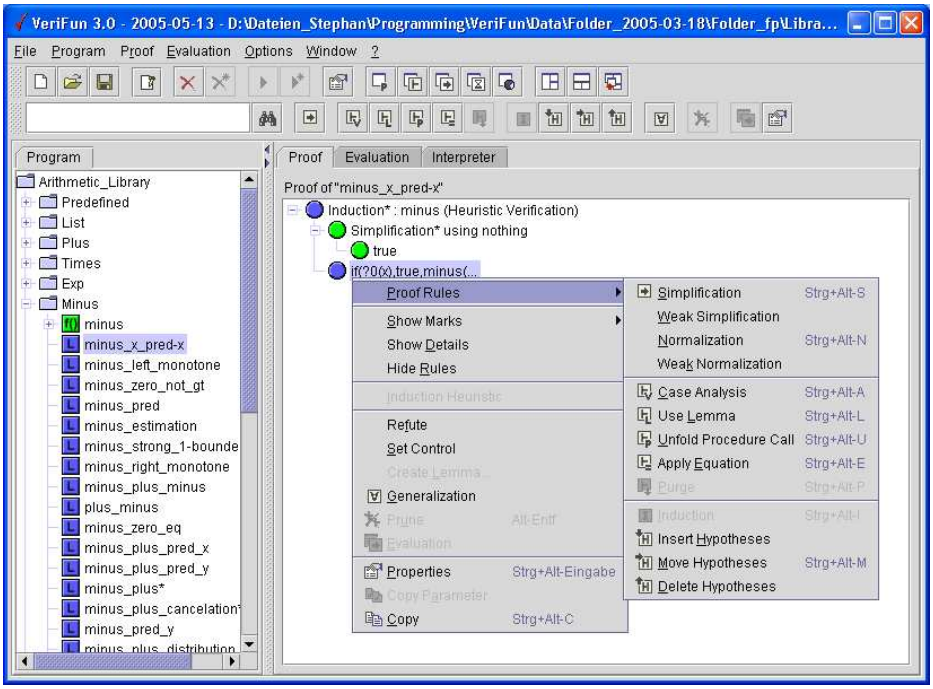


Abbildung 2.4: Das „*Proof Menu*“ zur Auswahl der Beweisregel.

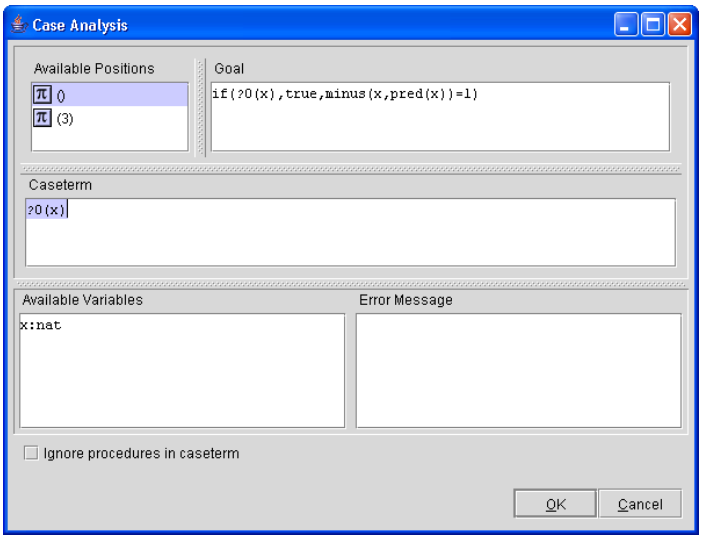


Abbildung 2.5: Eingabe-Dialog für die „*Case Analysis*“-Regel.

Beweisstatus	Programmstatus
proved	*verified*
completed	*developed*
unproved	*ready*
disproved	*falsified*
unprovable	*ready*

Prozeduren wird der Programmstatus auf ähnliche Weise zugewiesen. Sind alle Terminierungshypothesen der Prozedur verifiziert, so erhält die Prozedur den Programmstatus „*verified*“. Ist eine Terminierungshypothese noch nicht verifiziert, so wird der Programmstatus „*ready*“ zugewiesen. Konnten keine Terminierungshypothesen für die Prozedur erzeugt werden, so erhält die Prozedur den Status „*ignored*“. Typoperatoren erhalten immer den Programmstatus „*verified*“.

2.4 Beweisregeln des HPL-Kalküls

Die Beweisregeln des HPL-Kalküls werden durch Auswahl des entsprechenden Menüpunkts des Untermenüs **Proof Rules** auf die Sequenz des selektierten Knotens in einem Beweisbaum angewendet (Abbildung 2.4). Sie sind wie folgt definiert:

Case Analysis Die „*Case Analysis*“-Regel fügt eine Fallunterscheidung in einen Teilterm des Goal-Terms einer Sequenz ein. Der Benutzer spezifiziert den boolschen Term b bzgl. dessen die Fallunterscheidung definiert wird, sowie die Stelle π im Goal-Term, an der die Fallunterscheidung eingefügt werden soll (Abbildung 2.5). Weiter kann der Benutzer spezifizieren, ob die Prozeduren des Terms b durch eine mögliche spätere Anwendung der „*Computed Proof Rules*“ ausgewertet werden sollen oder nicht. Die „*Case Analysis*“-Regel erzeugt die folgenden Kindsequenzen:

$$\frac{H, IH \vdash g}{\begin{array}{l} H, IH \vdash g[\pi \leftarrow \texttt{if}(b, g|_\pi, \texttt{true})], \\ H, IH \vdash g[\pi \leftarrow \texttt{if}(b, \texttt{true}, g|_\pi)]. \end{array}}$$

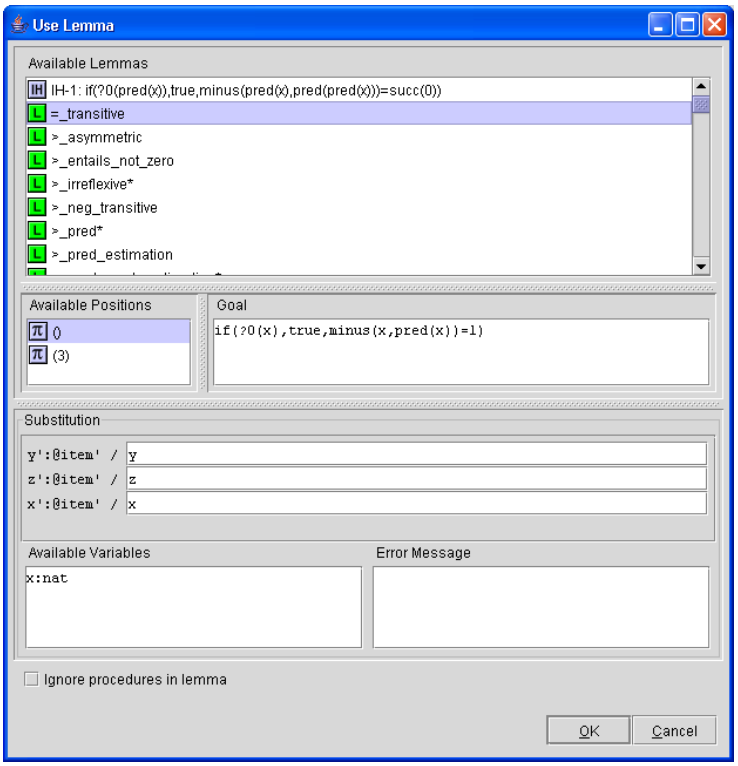


Abbildung 2.6: Eingabe-Dialog für die „*Use Lemma*“-Regel.

Spezifiziert der Benutzer einen nicht-boolschen Term b für die Fallunterscheidung, so wird eine *strukturelle* Fallunterscheidung durchgeführt:

$$\frac{H, IH \vdash g}{\begin{array}{c} H, IH \vdash g[\pi \leftarrow \texttt{if}(?cons_1(b), g|_\pi, \texttt{true})], \\ \ldots, \\ H, IH \vdash g[\pi \leftarrow \texttt{if}(?cons_n(b), g|_\pi, \texttt{true})]. \end{array}}$$

Use Lemma Diese Regel fügt eine Instanz eines Lemmas in den Goal-Term der aktuellen Sequenz ein. Der Benutzer wählt hierzu aus der Liste der Lemmata mit Programmstatus $\notin \{$„*ignored*“, „*falsified*“$\}$ ein Lemma

$$\texttt{lemma}\ name \Leftarrow \texttt{all}\ x_1 : \tau_1, \ldots, x_n : \tau_n\ body$$

aus und gibt eine Substitution σ zur Instantiierung der all-quantifizierten Variablen des Lemmas ein. Weiter wählt der Benutzer die Position π aus, an der die Instanz des Lemmas in den Goal-Term eingefügt werden soll (Abbildung 2.6):

$$\frac{H, IH \vdash g}{H, IH \vdash g[\pi \leftarrow \texttt{if}(\sigma(body), g|_\pi, \texttt{true})].}$$

Neben Instanzen von Lemmata kann der Benutzer mit dieser Regel auch Instanzen der Induktionshypothesen der aktuellen Sequenz in den Term einfügen. Weiter kann der Benutzer spezifizieren, ob die in der Instanz des Lemmas bzw. der Induktionshypothese enthaltenen Prozeduren von den „*Computed Proof Rules*“ ausgewertet werden sollen.

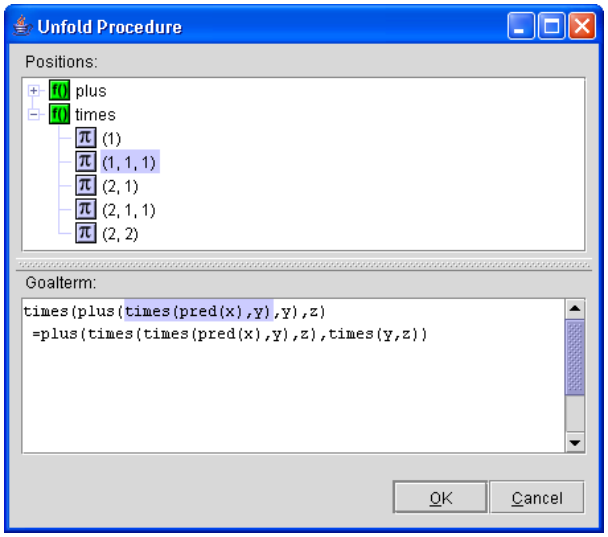


Abbildung 2.7: Eingabe-Dialog für die „*Unfold Procedure*"-Regel.

Unfold Procedure Mit Hilfe dieser Regel kann der Benutzer Prozeduraufrufe im Goal-Term durch die entsprechenden Instanzen der Prozedurrümpfe ersetzen. Der Benutzer wählt hierzu die Position π des Prozeduraufrufs $f(t_1, \ldots, t_n)$ im Goal-Term aus, der durch seinen Prozedurrumpf R_f^* ersetzt werden soll (Abbildung 2.7)[6]:

$$\frac{H, IH \vdash g}{H, IH \vdash g[\pi \leftarrow R_f^*[x_1/t_1, \ldots, x_n/t_n]].}$$

Apply Equation Mit Hilfe dieser Regel kann der Benutzer Teiltermersetzungen auf Basis von Gleichungen durchführen. So kann der Benutzer eine Gleichung $l{=}r$ der Hypothesenmenge H zur Ersetzung eines Teilterms l durch r an Position π verwenden:

$$\frac{H, IH \vdash g}{H, IH \vdash g[\pi \leftarrow r].}$$

In ähnlicher Weise kann der Benutzer eine Gleichung $l{=}r$ eines Teilterms $\texttt{if}(l{=}r, t_1, t_2)$ des Goal-Terms verwenden, um in t_1 einen Teilterm l durch r zu ersetzen. Die Gleichung $l{=}r$ wird dann als *lokale Gleichung* bezeichnet.

Weiter können Gleichungen, die durch Lemmata mit einem Programmstatus $\notin$ {„*ignored*", „*falsified*"} bzw. durch die Induktionshypothesen der Sequenz gegeben sind, angewendet werden. Hierzu repräsentiert ✓`eriFun` jedes Lemma und jede Induktionshypothese durch Klauseln. Der Benutzer kann dann eine Gleichung $l{=}r$ einer Klausel C eines Lemmas bzw. einer Induktionshypothese verwenden, um einen Teilterm $\sigma(l)$ des Goal-Terms an Position π zu ersetzen (Abbildung 2.8). Werden durch die Substitution σ nicht alle Variablen der Klausel instantiiert, so muss der Benutzer eine weitere Substitution θ spezifizieren, um auch diese Variablen zu instantiieren. Enthält $C' = C \setminus \{l{=}r\}$ weitere Terme, so ist die Teiltermersetzung von $\sigma(l)$ durch $\theta(\sigma(r))$ nur unter Voraussetzung der Gültigkeit des Terms $NOR(\theta(\sigma(C')))$ korrekt. Hierbei bezeichnet $NOR(\theta(\sigma(C')))$ die Konjunktion der Negation der Literale von $\theta(\sigma(C'))$. Die Teiltermersetzung muss daher durch diesen Term kontrolliert werden:

[6] Mit R_f^* bezeichnen wir den Prozedurrumpf der Prozedur f, wobei alle undefinierten Stellen $*$ durch rekursive Aufrufe der Form $f(x_1, \ldots, x_n)$ ersetzt sind. Auf die Notwendigkeit $*$ durch rekursive Aufrufe zu ersetzen gehen wir in Abschnitt 3.5 ein.

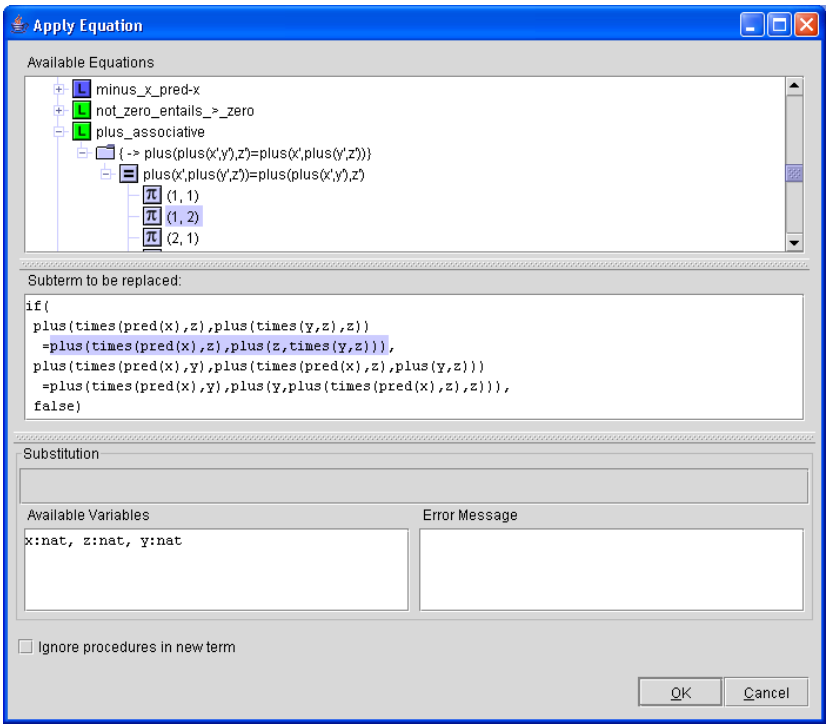


Abbildung 2.8: Eingabe-Dialog für die „*Apply Equation*"-Regel.

$$\frac{H, IH \vdash g}{H, IH \vdash g[\pi \leftarrow \texttt{if}(NOR(\theta(\sigma(C'))), \theta(\sigma(r)), g|_\pi)].}$$

Purge Diese Regel ermöglicht es, Variablenbindungen die durch Terme der Form `let` x`:=`t `in` r `end` eingeführt werden, in einen Teilterm des Terms r einzusetzen. Der Benutzer wählt hierzu die Position π des Teilterms des Goal-Terms aus, für den die Variablenbindungen eingesetzt werden sollen:

$$\frac{H, IH \vdash g}{H, IH \vdash g[\pi \leftarrow \delta_\pi(g|_\pi)].}$$

Hierbei bezeichnet δ_π die Substitution mit allen für den Teilterm $g|_\pi$ relevanten Variablenbindungen.

Induction Die „*Induction*"-Regel wendet ein Induktionsaxiom auf eine Sequenz der Form $\emptyset, \emptyset \vdash g$ an. Induktionsaxiome werden im ✓`eriFun`-System durch fundierte Relationenbeschreibungen repräsentiert [46]. Die zur Verfügung stehenden Relationenbeschreibungen werden aus den Typoperatordefinitionen und den Prozedurdefinitionen gewonnen. Aus diesen Relationenbeschreibungen wählt der Benutzer mittels des in Abbildung 2.9 dargestellten Dialogs die zu verwendende Relationenbeschreibung D aus. Weiter definiert der Benutzer eine Substitution σ, die die Variablen der Relationenbeschreibung R durch einige der im Goal-Term enthaltenen Variablen ersetzt. Diese Substitution legt somit festlegt über welche Variablen die Induktion angewendet werden soll:

$$\frac{\emptyset, \emptyset \vdash g}{H_1, \emptyset \vdash g, \ldots, H_n, \emptyset \vdash g, H_{n+1}, IH_1 \vdash g, \ldots, H_{n+m}, IH_m \vdash g.}$$

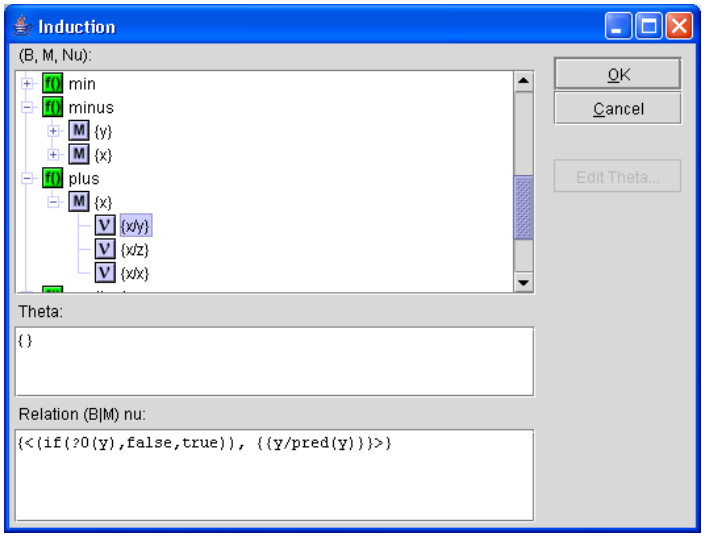


Abbildung 2.9: Eingabe-Dialog für die „*Induction*"-Regel.

Wie die Basisfälle

$$H_1, \emptyset \vdash g, \ldots, H_n, \emptyset \vdash g$$

und die Schrittfälle

$$H_{n+1}, IH_1 \vdash g, \ldots, H_{n+m}, IH_m \vdash g$$

einer Induktion in ✓eriFun auf Basis einer fundierten Relationenbeschreibung erzeugt werden, ist in Anhang B beschrieben.

Insert Hypotheses Diese Regel stellt dem Benutzer eine zweite Möglichkeit zur Fallunterscheidung zur Verfügung. Der Benutzer spezifiziert hierzu ein Atom b, bzgl. dessen die Fallunterscheidung definiert werden soll. Dieses Atom b bzw. seine Negation $\neg b$ wird dann in die Hypothesenmenge eingefügt:

$$\frac{H, IH \vdash g}{\begin{array}{l} H \cup \{b\}, IH \vdash g, \\ H \cup \{\neg b\}, IH \vdash g. \end{array}}$$

Wird vom Benutzer ein nicht-boolscher Term b zur Fallunterscheidung spezifiziert, so wird eine strukturelle Fallunterscheidung erzeugt:

$$\frac{H, IH \vdash g}{\begin{array}{l} H \cup \{\texttt{?}cons_1(b)\}, IH \vdash g, \\ \ldots, \\ H \cup \{\texttt{?}cons_n(b)\}, IH \vdash g. \end{array}}$$

Move Hypothesis Diese Regel verschiebt eine durch den Benutzer ausgewählte Hypothese $h \in H$ der Sequenz in den Goal-Term:

$$\frac{H, IH \vdash g}{H \setminus \{h\}, IH \vdash \texttt{if}(h, g, \texttt{true}).}$$

Eine Verschiebung einer Hypothese h ist notwendig, um geschachtelte Induktionen unter Berücksichtigung dieser Hypothese zu ermöglichen. Weiter ermöglicht diese Regel die symbolische Auswertung der Hypothese.

Delete Hypotheses Ist eine Hypothese $h \in H$ oder eine Induktionshypothese $ih \in IH$ für den weiteren Beweis irrelevant, so kann diese Hypothese bzw. Induktionshypothese mit Hilfe der Regel „*Delete Hypotheses*“ aus der Sequenz entfernt werden:

$$\frac{H, IH \vdash g}{H \setminus \{h\}, IH \vdash g,} \qquad \frac{H, IH \vdash g}{H, IH \setminus \{ih\} \vdash g.}$$

Die Anwendung dieser Regel ist notwendig, falls auf die Sequenz eine Induktion angewendet werden soll und die Hypothesen oder die Induktionshypothesen der Sequenzen nicht leer sind. Bei der Beweisregel handelt es sich um eine Generalisierung. Das bedeutet, dass die Ungültigkeit der Kindknoten der Sequenz nicht die Ungültigkeit der Sequenz implizieren.

Computed Proof Rules Die Beweisregeln, die unter dem Begriff „*Computed Proof Rules*“ zusammengefasst werden, werten den Goal-Term mit Hilfe des symbolischen Auswertungskalküls aus. Beim symbolischen Auswertungskalkül handelt es sich, wie bereits erwähnt um die vollautomatische Beweiskomponente des ✓`eriFun`-Systems. Das bedeutet, dass während der symbolischen Auswertung eines Goal-Terms keine Benutzerinteraktionen notwendig sind. Als „*Computed Proof Rules*“ werden die folgenden Beweisregeln bezeichnet:

- „*Simplification*“: Diese Beweisregel verwendet zur symbolischen Auswertung die Typoperator- und Prozedurdefinitionen, die durch den Lemma-Filter ausgewählten Lemmata sowie die Hypothesen und Induktionshypothesen der Sequenz.
- „*Weak Simplification*“: Die durch diese Regel definierte symbolische Auswertung unterscheidet sich von der symbolischen Auswertung mit „*Simplification*“ lediglich dahingehend, dass nur nicht-rekursiv-definierte Prozeduren ausgewertet werden.
- „*Normalization*“: Die „*Normalization*“-Regel schränkt die symbolische Auswertung dahingehend ein, dass keine Prozeduren ausgewertet werden. Das bedeutet, dass zur symbolischen Auswertung lediglich die Typoperatordefinitionen, die durch den Lemma-Filter ausgewählten Lemmata sowie die Hypothesen und Induktionshypothesen der Sequenz verwendet werden. Während der symbolischen Auswertung mit „*Normalization*“ wird im Vergleich zu „*Simplification*“ jedoch ein größer Suchraum für die Anwendung von Lemmata überprüft.
- „*Weak Normalization*“: Die durch diese Regel definierte symbolische Auswertung wertet weder Prozeduren aus, noch werden Lemmata zur Auswertung der Terme verwendet. Das bedeutet, dass die Regel lediglich einfache Umformungen durchführt.

Jede dieser Beweisregeln ist durch eine Ersetzung der Form

$$\frac{H, IH \vdash g}{H, IH \vdash g\Downarrow}$$

definiert, wobei $g\Downarrow$ den entsprechend vereinfachten Term bezeichnet.

Inconsistency Diese Beweisregel stellt eine Ausnahme unter den Beweisregeln des HPL-Kalküls dar, da sie nicht vom Benutzer explizit angewendet werden kann. Vielmehr wird sie von ✓`eriFun` automatisch nach Anwendung von „*Induction*“ oder „*Insert Hypotheses*“ auf die Kinderknoten angewendet, um zu überprüfen, ob die

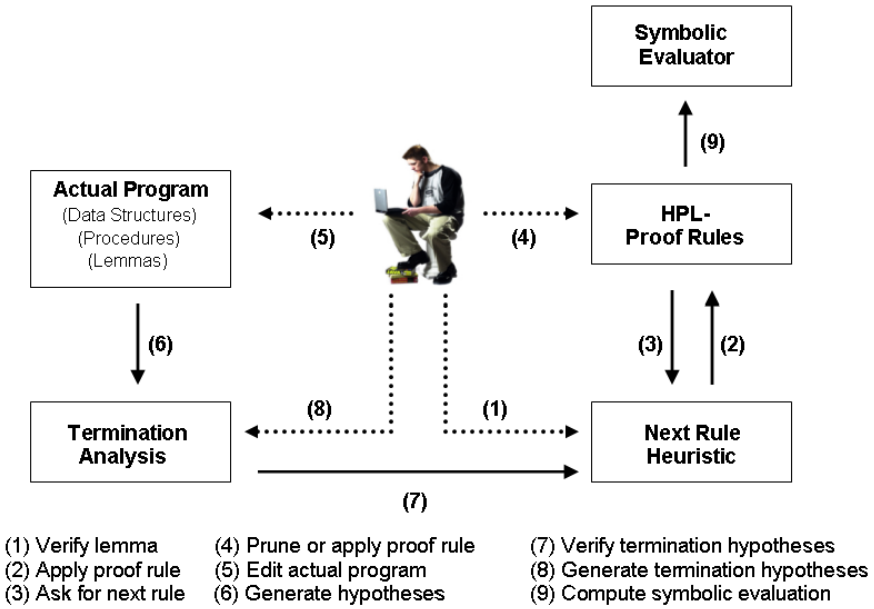


Abbildung 2.10: Interaktive (⇢) und automatisierte (→) Aktionen während der Arbeit mit √eriFun(siehe [58]).

Hypothesen der Kinderknoten widersprüchlich sind. Hierzu wird jede Hypothese $h \in H$ unter der Hypothesenmenge $H \backslash \{h\}$ symbolisch ausgewertet. Kann die Hypothese zu `false` ausgewertet werden, so ist die Hypothesenmenge widersprüchlich und die Sequenz damit gültig. Der Goal-Term kann daher durch `true` ersetzt werden:

$$\frac{H, IH \vdash g}{H, IH \vdash \texttt{true}.}$$

2.5 Semi-Automatische Verifikation

Neben denen durch den symbolischen Auswertungskalkül realisierten Automatisierungen existieren eine Reihe weiterer Automatisierungen, die den Benutzer bei der Beweiskonstruktion zusätzlich entlasten. Die generelle Arbeitsweise mit dem System sieht daher wie folgt aus.

Durch Anwenden des Menüpunkts **Program\Verify** auf ein Lemma mit Programmstatus „*ready*“, für das noch kein Beweisbaum konstruiert wurde, erzeugt das System zunächst einen so genannten Beweisplan zur Konstruktion eines initialen Beweisbaums (Aktion (1) in Abbildung 2.10). Dieser Beweisplan wird mit Hilfe der so genannten „*Next-Rule-Heuristic*“ generiert. Diese Heuristik entscheidet, welche der Beweisregeln des HPL-Kalküls am vielversprechendsten ist und wendet diese Regel auf die Blätter des aktuellen Beweisbaums an. Handelt es sich bei der ausgewählten Beweisregel um eine parametrisierte Beweisregel, so berechnet die „*Next-Rule-Heuristic*“ auf Basis weiterer Heuristiken die notwendigen Eingabeparameter. So wird beispielsweise für die „*Induction*“-Regel eine Relationenbeschreibung und eine Induktionsvariable automatisch ausgewählt, bzgl. derer die Induktion angewendet wird. Die „*Next-Rule-Heuristic*“ wendet auf die Blätter des aktuellen Beweisbaums solange neue HPL-Regeln an, bis entweder die Heuristik für das aktuelle Blatt keine Beweisregel mehr auswählt oder aber ein geschlossener Beweisbaum konstruiert wurde (Aktionen (2) und (3) in Abbildung 2.10). Wählt die Heuristik keine Beweisregel mehr aus, so muss der Benutzer unerwünschte Teile des Beweisbaums

abschneiden bzw. eine Beweisregel mit dem **Proof Rules** Menü auswählen, die auf eines der Blätter des Beweisbaums angewendet werden soll (Aktion (4) in Abbildung 2.10). Nach Anwendung der ausgewählten Regel übernimmt das System erneut die Kontrolle und wendet die „*Next-Rule-Heuristic*“ auf die neu erzeugten Blätter des Beweisbaums an (Aktionen (2) und (3) in Abbildung 2.10). HPL-Regeln, die aufgrund der „*Next-Rule-Heuristic*“ durch das ✓eriFun-System automatisch angewendet wurden, werden im Beweisbaum durch einen Stern markiert. Dadurch weiß der Benutzer, welche HPL-Regeln durch ihn und welche durch das System angewendet wurden.

Wird eine neue Prozedur in das aktuelle Programm eingefügt (Aktion (5) in Abbildung 2.10), so versucht das System die Terminierung dieser Prozedur automatisch nachzuweisen. Hierzu erzeugt das System Terminierungshypothesen für die Prozedur (Aktion (6) in Abbildung 2.10) und versucht für diese Terminierungshypothesen mit Hilfe der „*Next-Rule-Heuristic*“ jeweils einen geschlossenen Beweisbaum zu konstruieren (Aktion (7) in Abbildung 2.10). Können keine geschlossenen Beweisbäume erzeugt werden, so sind entsprechende Interaktionen des Benutzers erforderlich (Aktion (4) in Abbildung 2.10). Kann das System keine Terminierungshypothesen generieren, so muss der Benutzer durch Auswahl des Menüpunkts **Program\Set Termination** eine so genannte Terminierungsfunktion eingeben (Aktion (8) in Abbildung 2.10). Auf Basis dieser Terminierungsfunktion werden dann die Terminierungshypothesen der Prozedur erzeugt, auf die dann wieder die „*Next-Rule-Heuristic*“ angewendet wird (Aktion (7) in Abbildung 2.10).

2.6 Vollständiges Beispiel

Anhand eines Beispiels wollen wir in diesem Abschnitt zeigen, wie das ✓eriFun-System die Gültigkeit eines Lemmas mit Hilfe des symbolischen Auswertungskalküls nachweist.

Beispiel 2.1. Die Prozedur `mult` aus Abbildung 2.11 berechnet das Produkt seiner Argumente `x` und `y`. Hierzu überprüft die Prozedur für den Fall, dass `x` ungleich `0` gilt, mit Hilfe von `even`, ob `x` gerade ist. Ist dies der Fall, so ruft sich `mult` rekursive auf, wobei `x` für den rekursiven Aufruf mit `half` halbiert und `y` mit `dbl` verdoppelt wird. Ist `x` nicht gerade, so findet der gleiche rekursive Aufruf statt, wobei `y` mit Hilfe von `plus` auf das Ergebnis des rekursiven Aufrufs addiert wird.

Für die Prozedur `mult` wollen wir nun die Gültigkeit des folgenden Lemmas beweisen:

$$\texttt{lemma mult\_zero} \Leftarrow \texttt{all x : nat}\\ \texttt{if(?0(y), ?0(mult(x, y)), true)}\,.$$

Zum Beweis dieses Lemmas erzeugt ✓eriFun den folgenden initialen Knoten:

$$\emptyset, \emptyset \vdash \texttt{if(?0(y), ?0(mult(x, y)), true)}$$

Die „*Next-Rule-Heuristic*“ schlägt dann eine Induktion über die Relationenbeschreibung der Prozedur `mult` vor. Durch diese Induktion werden die beiden folgenden Sequenzen als Induktionsbasis und Induktionsschritt erzeugt:

$$\{\texttt{?0(x)}\}, \emptyset \vdash \texttt{if(?0(y), ?0(mult(x, y)), true)}, \tag{2.4}$$

$$\{\texttt{?succ(x)}\}, \{\texttt{all y : nat if(?0(y), ?0(mult(half(x), y)), true)}\} \vdash \\ \texttt{if(?0(y), ?0(mult(x, y)), true)}. \tag{2.5}$$

Die Gültigkeit dieser Sequenzen und damit die Gültigkeit des Lemmas wird von ✓eriFun durch symbolische Auswertung der Goal-Terme bewiesen. Hierzu wendet ✓eriFun mit Hilfe der „*Next-Rule-Heuristic*“ auf beide Sequenzen die Regel

```
function plus(x : nat, y : nat) : nat ⇐
  if(?0(x),
     y,
     succ(plus(pred(x), y)))

function dbl(x : nat) : nat ⇐
  if(?0(x),
     0,
     succ(succ(dbl(pred(x)))))

function even(x : nat) : nat ⇐
  if(?0(x),
     true,
     if(?0(pred(x)),
        false,
        even(pred(pred(x)))))

function half(x : nat) : nat ⇐
  if(?0(x),
     0,
     if(?0(pred(x)),
        0,
        succ(half(pred(pred(x))))))

function mult(x : nat, y : nat) : nat ⇐
  if(?0(x),
     0,
     if(even(x),
        mult(half(x), dbl(y)),
        plus(mult(half(x), dbl(y)), y))),
```

Abbildung 2.11: Prozedurdefinitionen für `plus`, `dbl`, `half`, `even` und `mult`.

$$
\begin{aligned}
&\texttt{if}(\texttt{?0}(\texttt{y}), \texttt{?0}(\underline{\texttt{mult}(\texttt{x},\texttt{y})}), \texttt{true}) && \xrightarrow{(1)}\\
&\texttt{if}(\texttt{?0}(\texttt{y}), \texttt{?0}(\texttt{if}(\underline{\texttt{?0}(\texttt{x})}, \texttt{0}, \ldots)), \texttt{true}) && \xrightarrow{(2)}\\
&\texttt{if}(\texttt{?0}(\texttt{y}), \texttt{?0}(\underline{\texttt{if}(\texttt{true}, \texttt{0}, \ldots)}), \texttt{true}) && \xrightarrow{(3)}\\
&\texttt{if}(\texttt{?0}(\texttt{y}), \texttt{?0}(\texttt{0}), \texttt{true}) && \xrightarrow{(4)}\\
&\texttt{if}(\texttt{?0}(\texttt{y}), \texttt{true}, \texttt{true}) && \xrightarrow{(5)}\\
&\texttt{true}
\end{aligned}
$$

Abbildung 2.12: Symbolische Auswertung des Goal-Terms der Sequenz (2.4). Die Teilterme, die sich in den Auswertungsschritten ändern, sind unterstrichen. Weiterhin sind die einzelnen Auswertungsschritte durchnummeriert.

if(?0(y), ?0(mult(x, y)), true)	(1)	if(?0(y), ?0(if(even(x), 0, y)), true)
if(?0(y), ?0(if(?0(x), 0, if(even(x), mult(half(x), dbl(y)), plus(mult(half(x), dbl(y)), y)))), true)	(2)	if(?0(y), if(even(x), ?0(0), ?0(y)), true)
if(?0(y), ?0(if(false, 0, if(even(x), mult(half(x), dbl(y)), plus(mult(half(x), dbl(y)), y)))), true)	(3)	if(?0(y), if(even(x), true, ?0(y)), true)
if(?0(y), ?0(if(even(x), mult(half(x), dbl(y)), plus(mult(half(x), dbl(y)), y))), true)	(4)	if(?0(y), if(even(x), true, ?0(y)), true)
if(?0(y), ?0(if(even(x), mult(half(x), if(?0(y), 0, succ(succ(dbl(pred(y))))), plus(mult(half(x), dbl(y)), y)))), true)	(5)	if(?0(y), if(even(x), true, true), true)
if(?0(y), ?0(if(even(x), mult(half(x), if(true, 0, succ(succ(dbl(pred(y)))))), plus(mult(half(x), dbl(y)), y))), true)	(6)	if(?0(y), true, true)
if(?0(y), ?0(if(even(x), mult(half(x), 0), plus(mult(half(x), dbl(y)), y))), true)	(7)	true
if(?0(y), ?0(if(even(x), if(?0(0), 0, mult(half(x), 0)), plus(mult(half(x), dbl(y)), y))), true)	(8)	
if(?0(y), ?0(if(even(x), if(true, 0, mult(half(x), 0)), plus(mult(half(x), dbl(y)), y))), true)	(9)	
if(?0(y), ?0(if(even(x), 0, plus(mult(half(x), dbl(y)), y))), true)	(10)	
if(?0(y), ?0(if(even(x), 0, plus(mult(half(x), if(?0(y), 0, succ(succ(dbl(pred(y)))))), y))), true)	(11)	
if(?0(y), ?0(if(even(x), 0, plus(mult(half(x), if(true, 0, succ(dbl(pred(y))))), y))), true)	(12)	
if(?0(y), ?0(if(even(x), 0, plus(if(?0(0), 0, mult(half(x), 0)), y))), true)	(13)	
if(?0(y), ?0(if(even(x), 0, plus(mult(half(x), 0), y))), true)	(14)	
if(?0(y), ?0(if(even(x), 0, plus(if(?0(0), 0, mult(half(x), 0)), y))), true)	(15)	
if(?0(y), ?0(if(even(x), 0, plus(if(true, 0, mult(half(x), 0)), y))), true)	(16)	
if(?0(y), ?0(if(even(x), 0, plus(0, y))), true)	(17)	
if(?0(y), ?0(if(even(x), 0, if(?0(0), y, succ(plus(pred(0), y))))), true)	(18)	
if(?0(y), ?0(if(even(x), 0, if(true, y, succ(plus(pred(0), y))))), true)	(19)	

Abbildung 2.13: Symbolische Auswertung des Goal-Terms der Sequenz (2.5). Die Teilterme, die sich in den Auswertungsschritten ändern, sind unterstrichen. Weiterhin sind die einzelnen Auswertungsschritte durchnummeriert.

„*Simplification*“ an. Für die Sequenz (2.4) ergibt sich dabei die in Abbildung 2.12 dargestellte symbolische Auswertung. Im ersten Auswertungsschritt dieser symbolischen Auswertung wird der Prozeduraufruf `mult(x,y)` durch den entsprechend instantiierten Prozedurrumpf der Prozedur ersetzt. Anschließend wird aufgrund der Hypothesenmenge der Sequenz (2.4) die Bedingung `?0(x)` des instantiierten Prozedurrumpfs zu `true` ausgewertet. In den Schritten (3)-(5) kann dann der Term zu `true` vereinfacht werden. Damit ist die Gültigkeit der Sequenz (2.4) bewiesen.

Die symbolische Auswertung des Goal-Terms der Sequenz (2.5) ist in Abbildung 2.13 dargestellt. Im ersten Auswertungsschritt der symbolischen Auswertung wird wieder der Prozeduraufruf `mult(x,y)` durch den entsprechend instantiierten Prozedurrumpf der Prozedur ersetzt. Anschließend wird aufgrund der Hypothesenmenge der Sequenz (2.5) die Bedingung `?0(x)` des instantiierten Prozedurrumpfs zu `false` ausgewertet und in Auswertungsschritt (3) der `if`-Term auf den entsprechenden `else`-Teil reduziert. In den Auswertungsschritten (4)-(6) wird dann der Prozeduraufruf `dbl(y)` zu `0` ausgewertet und in den Auswertungsschritten (7)-(9) die Induktionshypothese

$$\texttt{if}(\texttt{?0}(\texttt{y}), \texttt{?0}(\texttt{mult}(\texttt{half}(\texttt{x}), \texttt{y})), \texttt{true})$$

als bedingte Gleichung der Form

$$\texttt{?0}(\texttt{y}) \Longrightarrow \texttt{mult}(\texttt{half}(\texttt{x}), \texttt{y})\texttt{=0}$$

verwendet. Diese Auswertungen, d.h die Auswertung des Prozeduraufrufs `dbl(y)` und die Verwendung der Induktionshypothese als bedingte Gleichung, wiederholen sich in den Auswertungsschritten (10)-(15). In den Auswertungsschritten (16)-(18) wird anschließend der Prozeduraufruf `plus(0,y)` zu `y` ausgewertet und in Auswertungsschritt (19) das Funktionssymbol `?0` in den `then-` und `else`-Teil des `if`-Terms `if(even(x),0,y)` hineingezogen. Abschließend wird in den Auswertungsschritten (20)-(24) der verbleibende Term zu `true` vereinfacht. Damit ist auch die Gültigkeit der Sequenz (2.5) und somit insgesamt die Gültigkeit des Lemmas `mult_zero` bewiesen. □

Teil I

Formale Grundlagen

Kapitel 3

Die Programmiersprache $\mathcal{L}$

Wir definieren in diesem Kapitel die Syntax und Semantik der in ✓`eriFun` verwendeten Programmiersprache $\mathcal{L}$. Hierbei handelt es sich, wie bereits beschrieben, um eine polymorph getypte, funktionale Programmiersprache ohne Prozeduren höherer Ordnung. Programme der Sprache $\mathcal{L}$ bestehen aus Typoperator- und Prozedurdefinitionen. Die Typoperatordefinitionen führen hierbei neue freie algebraische Datenstrukturen in ein Programm ein und die Prozedurdefinitionen neue Algorithmen. Die durch Typoperator- und Prozedurdefinitionen eingeführten Funktionssymbole sind dabei nicht notwendigerweise vollständig-definiert. Beispielsweise führt die Typoperatordefinition

```
structure list ⇐
  empty, add(hd : nat, tl : list)
```
(3.1)

die Funktionssymbole `hd` und `tl` als Selektoren des Konstruktors `add` ein. Dabei wird die Bedeutung der Terme `hd`(`add`(t_1, t_2)) und `tl`(`add`(t_1, t_2)) für beliebige Terme t_1 und t_2 eines terminierenden Programms festgelegt, jedoch nicht die Bedeutung der Terme `hd`(`empty`) und `tl`(`empty`). Die Selektoren `hd` und `tl` sind also unvollständig-definiert. Ein anderes Beispiel für unvollständig-definierte Funktionssymbole ist durch die folgende Prozedurdefinition gegeben:

```
function last(l : list) : nat ⇐
  if(?empty(t),
     *,
     if(?empty(tl(l)),
        hd(l),
        last(tl(l))))
```

Durch diese Prozedurdefinition ist zwar der Wert des Terms `last`(`add`(t_1, t_2)) für beliebige Terme t_1 und t_2 eines terminierenden Programms festgelegt, jedoch nicht der Wert des Terms `last`(`empty`). Unvollständig-definierte Funktionssymbole werden auch als „*underspecification*“ [23] oder als „*loose specification*“ [30] bezeichnet.

Die Semantik unvollständig-definierter Funktionssymbole wird in funktionalen Sprachen üblicherweise mit Hilfe spezieller Fehlerwerte wie beispielsweise `undef` definiert. Das bedeutet, dass ein Term t durch den Wert `undef` gedeutet wird, falls die Bedeutung des Terms aufgrund der unvollständig-definierten Funktionssymbole nicht festgelegt ist. Der Term

```
last(empty)=last(empty)
```

wird dann beispielsweise durch den Wert `undef` gedeutet, da die Semantik des Teilterms `last`(`empty`) durch die Prozedurdefinition von `last` nicht gegeben ist. Für

die Verifikation von Programmen hat die Verwendung von Fehlerwerten wie `undef` den Nachteil, dass die zur Verifikation verwendete Logik die Fehlerwerte speziell behandeln muss und die Logik dementsprechend kompliziert ist.

Die Semantik unvollständig-definierter Funktionssymbole in $\mathcal{L}$ basiert nicht auf speziellen Fehlerwerten. Vielmehr wird ein Term wie `last(empty)` durch einen beliebigen aber unbekannten Wert des Wertebereichs des Funktionssymbols `last` gedeutet. Das führt dazu, dass wir eine Gleichung wie beispielsweise

```
last(empty)=last(empty)
```

als gültig deuten können. Für die Verifikation von Programmen hat diese Semantik den Vorteil, dass die zur Verifikation verwendete Logik recht einfach ist, da die spezielle Behandlung von Fehlerwerten entfällt. Nachteil der Semantik ist, dass jeder Interpreter der Sprache unvollständig ist. Das bedeutet, dass ein Interpreter nicht für alle Terme die Deutung der Terme berechnen kann. So wird beispielsweise jeder Interpreter die Auswertung des Terms `last(empty)` abbrechen, da diesem Term keine *eindeutige* Semantik zugeordnet werden kann.

In [59] wird für eine nicht-polymorph getypte Version der Sprache $\mathcal{L}$ die Semantik von Programmen mit unvollständig-definierten Funktionssymbolen auf Basis einer operationalen Semantik mit Hilfe so genannter fairer Vervollständigungen definiert. Hierzu werden die Typoperator- und die Prozedurdefinitionen durch Angabe so genannter Beispielterme syntaktisch vervollständigt. Das bedeutet, dass eine faire Vervollständigung wieder ein Programm der Sprache ist. Jede der fairen Vervollständigungen definiert eine zulässige Interpretation der Funktionssymbole. Die einzige Einschränkung für die Vervollständigungen ist dabei, dass das Terminierungsverhalten des Programms nicht beeinflussen werden darf. Für die Typoperatordefinition (3.1) ist beispielsweise

```
structure list ⇐
  empty, add(hd : nat/0, tl : list/empty)
```

eine mögliche faire Vervollständigung. Diese Vervollständigung legt als Ergebnis des Terms `hd(empty)` den Wert `0` fest und als Ergebnis des Terms `tl(empty)` den Wert `empty`. Eine Aussage über ein terminierendes Programm P mit unvollständig-definierten Funktionssymbolen ist dann gültig, falls die Aussage in jeder fairen Vervollständigung des Programms P gültig ist. Die Methode der fairen Vervollständigungen ist insbesondere deshalb attraktiv, da sie auf einfache Weise die Semantik unvollständig-definierter Programme auf Basis vollständig-definierter Programme definiert.

Im Gegensatz zu [59] können wir für unsere polymorph-getypte Version der Sprache $\mathcal{L}$ unvollständig-definierte Programm nicht durch syntaktische Erweiterungen vervollständigen. Grund hierfür ist, dass nicht für jeden Typ einer polymorph-getypten Programmiersprache ohne weiteres geeignete Beispielterme angegeben werden können. So existiert beispielsweise für den polymorphen Typ `@a` kein Konstruktorgrundterm.[1] Wir modifizieren daher die Idee der fairen Vervollständigungen und definieren die Semantik unserer Programmiersprache auf Basis so genannter P-Interpretationen. Diese P-Interpretationen deuten alle Ausdrücke, deren Deutung durch die unvollständige Definition eines Programms P offen geblieben sind. Es ist zu beachten, dass es sich bei P-Interpretationen nicht um eine syntaktische Erweiterung des Programms P handelt, sondern um ein Konzept der Metasprache. Durch die Verwendung von P-Interpretationen gelingt es uns die Einfachheit des Ansatzes der fairen Vervollständigungen auf polymorph-getypte Programmiersprachen zu übertragen und so eine leicht verständliche Semantik der Sprache $\mathcal{L}$ zu definieren.

[1] `@a` bezeichnet in $\mathcal{L}$ eine Typvariable. Der Typ @a repräsentiert somit jeden beliebigen Typ.

Das Kapitel gliedert sich wie folgt. In Abschnitt 3.1 treffen wir einige grundlegende Vereinbarungen für die nachfolgenden Definitionen. In Abschnitt 3.2 führen wir dann Typen und Terme ein. Die Syntax der Sprache $\mathcal{L}$ legen wir durch die Typoperator- und Prozedurdefinitionen in Abschnitt 3.3 fest. In Abschnitt 3.4 definieren wir die Semantik von $\mathcal{L}$-Programmen. Danach geben wir in Abschnitt 3.5 einige grundlegende Eigenschaften der durch ein Programm eingeführten Funktionssymbole an. In Abschnitt 3.6 gehen wir dann auf einige Fragen bzgl. unvollständiger Fallunterscheidungen ein. Abschließend führen wir in Abschnitt 3.7 noch einige syntaktische Beschränkungen für Programme ein, um die Definitionen der nachfolgenden Kapitel zu vereinfachen.

Hinweis: Einige der verwendeten mathematischen Begriffe und Notationen sind in Anhang A definiert. Insbesondere die Schreibweisen für Listen, partielle Abbildungen und Äquivalenzrelationen sind dort aufgeführt. Der Leser ist aufgefordert bei Bedarf im Anhang nachzuschlagen.

3.1 Grundlegende Vereinbarungen

Um die Lesbarkeit der nachfolgenden Definitionen zu verbessern, setzen wir für dieses und alle nachfolgenden Kapitel die folgenden abzählbar unendlichen und paarweise disjunkten Mengen voraus:

- $\mathcal{V}$ die Menge der ungetypten Termvariablen.
- $\mathcal{S}$ die Menge der Funktionssymbole.
- $\mathcal{V}_{\mathcal{T}yp}$ die Menge der Typvariablen.
- $\mathcal{S}_{\mathcal{T}yp}$ die Menge der Typoperatoren.
- $\mathcal{S}_{\mathcal{L}em}$ die Menge der Lemmabezeichner.

Für die im Folgenden verwendeten konkreten Symbole, wie z.B. `bool`, `nat`, `succ`, `pred`, etc. setzen wir voraus, dass diese Symbole in den entsprechenden Mengen enthalten sind.

3.2 Typen und Terme

Wir bezeichnen eine partielle Abbildung $\Omega : \mathcal{S}_{\mathcal{T}yp} \mapsto \mathbb{N}$ als *Typsignatur* und nennen für ein $\zeta \in dom(\Omega)$ das Bild $\Omega(\zeta)$ *Stelligkeit von* ζ bzgl. der Typsignatur Ω. Statt $\zeta \in dom(\Omega)$ schreiben wir auch kurz $\zeta \in \Omega$. Aufbauend auf Typsignaturen definieren wir Typen.

Definition 3.1 (Typen)**.** Sei Ω eine Typsignatur. Die Menge $\mathcal{T}yp(\Omega)$ der *Typen über* Ω ist die kleinste Menge mit

(1) $\nu \in \mathcal{T}yp(\Omega)$ falls $\nu \in \mathcal{V}_{\mathcal{T}yp}$, und

(2) $\zeta[\tau_1, \ldots, \tau_n] \in \mathcal{T}yp(\Omega)$ falls $\Omega(\zeta) = n$ und $\tau_1, \ldots, \tau_n \in \mathcal{T}yp(\Omega)$.

□

Die Menge der Typvariablen eines Typs τ notieren wir mit $tv(\tau)$ und die Teiltyprelation mit $\geq_{\mathcal{T}yp}$. Enthält τ keine Typvariablen, so bezeichnen wir τ als *monomorphen* Typ, andernfalls als *polymorphen* Typ. Die Menge der monomorphen Typen über Ω notieren wir mit $\mathcal{T}yp^{\text{mono}}(\Omega)$. Weiter nennen wir eine Abbildung $\xi : \mathcal{V}_{\mathcal{T}yp} \to \mathcal{T}yp(\Omega)$ mit $\xi(\nu) \neq \nu$ für nur endlich viele $\nu \in \mathcal{V}_{\mathcal{T}yp}$ eine *Typsubstitution*

über Ω und bezeichnen die Menge aller Typsubstitutionen über Ω mit $\mathcal{Subst}(\Omega)$. Für eine Typsubstitution ξ heißt dann die Menge

$$dom(\xi) := \{\nu \in \mathcal{V}_{\mathcal{T}yp} \mid \xi(\nu) \neq \nu\}$$

Domain von ξ und die Menge

$$rg(\xi) := \{\xi(\nu) \in \mathcal{T}yp(\Omega) \mid \nu \in dom(\xi)\}$$

Range von ξ. Gilt $dom(\xi) = \{\nu_1, \ldots, \nu_n\}$, so notieren wir die Typsubstitution ξ mit $\{\nu_1/\xi(\nu_1), \ldots, \nu_n/\xi(\nu_n)\}$. Die leere Typsubstitution $\{\}$ notieren wir auch mit ϵ. Weiter schreiben wir $\xi \subseteq_{dom} \xi'$, falls $dom(\xi) \subseteq dom(\xi')$ gilt. Für eine Menge V von Typvariablen bezeichnen wir mit $\mathcal{Subst}_V(\Omega)$ die Menge aller Typsubstitutionen ξ mit $dom(\xi) = V$. Für eine Typsubstitution ξ und eine Menge V definieren wir die auf V beschränkte Typsubstitution $\xi|_V$ durch

$$\begin{array}{lcll} \xi|_V(x) & := & \xi(x) & \text{falls } x \in V, \\ \xi|_V(x) & := & x & \text{sonst.} \end{array}$$

Die homomorphe Erweiterung einer Typsubstitution ξ auf Typen definieren wir schließlich durch

$$\xi(\zeta[\tau_1, \ldots, \tau_n]) \quad := \quad \zeta[\xi(\tau_1), \ldots, \xi(\tau_n)].$$

Gilt $dom(\xi) = \{\nu_1, \ldots, \nu_n\}$, so schreiben wir für $\xi(\tau)$ auch $\tau[\nu_1/\xi(\nu_1), \ldots, \nu_n/\xi(\nu_n)]$.

Nachfolgend bezeichnen wir mit $\nu, \nu', \nu_1, \nu_2, \ldots$ immer Typvariablen, mit ζ, ζ', $\zeta_1, \zeta_2, \ldots$ Typoperatoren, mit $\tau, \tau', \tau_1, \tau_2, \ldots$ Typen und mit $\xi, \xi', \xi_1, \xi_2, \ldots$ Typsubstitutionen. Weiter gehen wir im Folgenden davon aus, dass konkrete Typvariablen immer mit dem Zeichen `@` beginnen.

Beispiel 3.2. Sei $\Omega = \{\texttt{nat} : 0, \texttt{list} : 1\}$ eine Typsignatur. Seien weiter `@a`, `@b` Typvariablen. Die Ausdrücke

$$\texttt{@a},\ \texttt{nat},\ \texttt{list[@b]}$$

sind dann Typen über der Typsignatur Ω und die Abbildungen

$$\begin{array}{lcl} \xi_1 & = & \{\texttt{@a/list[@b]}\}, \\ \xi_2 & = & \{\texttt{@a/nat}, \texttt{@b/list[@b]}\} \end{array}$$

sind Typsubstitutionen. Die Domains und Ranges dieser Typsubstitutionen sind wie folgt gegeben:

$$\begin{array}{rcl} dom(\xi_1) & = & \{\texttt{@a}\}, \\ dom(\xi_2) & = & \{\texttt{@a}, \texttt{@b}\}, \\ \\ rg(\xi_1) & = & \{\texttt{list[@b]}\}, \\ rg(\xi_2) & = & \{\texttt{nat}, \texttt{list[@b]}\}. \end{array}$$

Weiterhin gilt:

$$\begin{array}{rcl} \xi_1(\texttt{list[@b]}) & = & \texttt{list[@b]}, \\ \xi_2(\texttt{list[@b]}) & = & \texttt{list[list[@b]]}. \end{array}$$

□

Kommen wir nun zur Definition von Termen. Terme sind in der Sprache $\mathcal{L}$ explizit-getypt. Das bedeutet, dass für jeden Term und jeden Teilterm eines Terms explizit der Typ angegeben werden muss. Unsere Definition explizit-getypter Terme orientiert sich hierbei an [2]. Es sei an dieser Stelle darauf hingewiesen, dass die

Verwendung explizit-getypter Terme die Ausdrucksstärke der Sprache $\mathcal{L}$ nicht einschränkt, jedoch den Vorteil hat, dass die in den nachfolgenden Kapiteln definierte symbolische Auswertung stabil bzgl. Typisierung der Terme ist.[2] Zur Definition von Termen führen wir zunächst Ω-*getypte Termvariablen* ein.

Definition 3.3 (Termvariablen)**.** Sei Ω eine Typsignatur. Ein Ausdruck der Form x_τ mit $x \in \mathcal{V}$ und $\tau \in \mathcal{T}yp(\Omega)$ heißt Ω-*getypte Termvariable* (oder kurz *Termvariable*). Die Menge aller Ω-getypten Termvariablen wird mit $\mathcal{V}(\Omega)$ und die Menge aller Ω-getypten Termvariablen vom Typ τ mit $\mathcal{V}_\tau(\Omega)$ bezeichnet. Eine Ω-getypte Termvariable x_τ heißt *monomorph getypt*, falls τ ein monomorpher Typ ist. Die Menge aller monomorph getypten Termvariablen über Ω wird mit $\mathcal{V}^{\text{mono}}(\Omega)$ bezeichnet. □

Es ist zu beachten, dass x_τ und $x_{\tau'}$ verschiedene Termvariablen bezeichnen, falls $\tau \neq \tau'$ gilt. Für eine Liste $x^1_{\tau_1}, \ldots, x^n_{\tau_n}$ von Termvariablen schreiben wir häufig auch kurz $x^*_{\tau^*}$, wobei x^* die Liste $x^1, \ldots, x^n$ bezeichnet und τ^* die Liste $\tau_1, \ldots, \tau_n$. Ist der Typ τ einer Termvariable x_τ irrelevant oder aus dem Zusammenhang klar, so schreiben wir für die Termvariable auch kurz x. Gelegentlich ist es notwendig, für eine endliche Menge V von Termvariablen eine neue Termvariable x_τ zu erzeugen, für die $x_\tau \notin V$ gilt. Wir definieren hierzu die folgende Funktion:

$$Var_\tau(V) \quad := \quad \varepsilon_0(\mathcal{V}_\tau(\Omega) \setminus V).^3$$

Für eine Typsignatur Ω bezeichnen wir eine partielle Abbildung $\Sigma : \mathcal{S} \mapsto \mathcal{T}yp(\Omega)^+$ als Ω-*Termsignatur* oder kurz *Termsignatur* und nennen für ein $f \in dom(\Sigma)$ das Bild $\Sigma(f)$ *Stelligkeit* von f bzgl. der Termsignatur Σ. Eine Stelligkeit $\langle \tau_1, \ldots, \tau_n, \tau_{n+1} \rangle$ notieren wir häufig auch durch $\tau_1 \times \ldots \times \tau_n \to \tau_{n+1}$. Das Paar $\langle \Omega, \Sigma \rangle$ nennen wir *Signatur*. Aufbauend auf Signaturen definieren wir Terme.

Definition 3.4 (Terme)**.** Sei $\langle \Omega, \Sigma \rangle$ eine Signatur. Für einen Typ $\tau \in \mathcal{T}yp(\Omega)$ ist die Menge $\mathcal{T}erm_\tau(\Omega, \Sigma)$ der Terme von Typ τ über $\langle \Omega, \Sigma \rangle$ definiert als die kleinste Menge mit

(1) $x_\tau \in \mathcal{T}erm_\tau(\Omega, \Sigma)$, falls $x_\tau \in \mathcal{V}(\Omega)$,

(2) $f_\tau(t_1, \ldots, t_n) \in \mathcal{T}erm_\tau(\Omega, \Sigma)$, falls $t_1 \in \mathcal{T}erm_{\tau_1}(\Omega, \Sigma), \ldots, t_n \in \mathcal{T}erm_{\tau_n}(\Omega, \Sigma)$, $\Sigma(f) = \tau'_1 \times \ldots \times \tau'_n \to \tau'$ und eine Typsubstitution $\xi \in \mathcal{S}ubst(\Omega)$ existiert mit $\tau = \xi(\tau')$ und $\tau_1 = \xi(\tau'_1), \ldots, \tau_n = \xi(\tau'_n)$,

(3) `let` $x_{\tau'}$`:=`t `in` r `end` $\in \mathcal{T}erm_\tau(\Omega, \Sigma)$, falls $x_{\tau'} \in \mathcal{V}(\Omega)$, $t \in \mathcal{T}erm_{\tau'}(\Omega, \Sigma)$ und $r \in \mathcal{T}erm_\tau(\Omega, \Sigma)$.

Die Menge aller Terme über $\langle \Omega, \Sigma \rangle$ wird mit $\mathcal{T}erm(\Omega, \Sigma)$ bezeichnet. □

Ist der Typ τ irrelevant oder aus dem Zusammenhang klar, so schreiben wir für einen Term $f_\tau(t_1, \ldots, t_n)$ kurz $f(t_1, \ldots, t_n)$. Für einen Term $t = f_{\tau_{n+1}}(t_1, \ldots, t_n)$ mit $\Sigma(f) = \tau'_1 \times \ldots \times \tau'_n \to \tau'_{n+1}$ und $t_i \in \mathcal{T}erm_{\tau_i}(\Omega, \Sigma)$ für alle $i \in \{1, \ldots, n\}$ notieren wir mit ξ_t die $\subseteq_{dom}$-minimale Typsubstitution mit $\xi_t(\tau'_i) = \tau_i$ für alle

[2] Stabil bzgl. Typisierung bedeutet, dass ein Term durch die symbolische Auswertung seinen Typ nicht verändert. Beispielsweise wird der explizit-getypte Term $\mathtt{tl}_{\mathtt{list[nat]}}(\mathtt{add}_{\mathtt{list[nat]}}(1_{\mathtt{nat}}, \mathtt{empty}_{\mathtt{list[nat]}}))$ durch die symbolische Auswertung zu $\mathtt{empty}_{\mathtt{list[nat]}}$ vereinfacht. Der Typ `list[nat]` des Terms bleibt also unverändert. In einer Sprache mit implizit-getypten Termen würde dem Term `tl(add(1, empty))` der Typ `list[nat]` zugewiesen. Der Term würde dann durch die symbolische Auswertung zu `empty` vereinfacht und anschließend dem Term `empty` der Typ `list[@a]` zugewiesen. Das bedeutet, dass die symbolische Auswertung den Typ eines implizit-getypten Terms modifiziert.

[3] ε_0 bezeichnet die Auswahlfunktion. Siehe hierzu Anhang A.

$i \in \{1, \ldots, n+1\}$.[4] Weiter bezeichnen wir einen Term der Form `let` x`:=`t `in` r `end` als `let`-*Term*. Den Term r nennen wir dann Rumpf des `let`-Terms und den Teilausdruck x`:=`t Variablenbindung des `let`-Terms. Die Menge der *freien bzw. gebundenen Termvariablen* eines Terms t ist wie üblich definiert und wird mit $fv(t)$ bzw. $bv(t)$ bezeichnet. Die Teiltermrelation notieren wir mit $\geq_{\mathcal{T}erm}$ und bezeichnen einen Term t als `let`-*frei*, falls t keinen `let`-Term als Teilterm enthält. Einen Term t mit $fv(t) = \emptyset$ nennen wir *Grundterm*. Die Menge aller Typvariablen eines Terms t bezeichnen wir mit $tv(t)$. Die homomorphe Erweiterung einer Typsubstitution ξ auf Terme definieren wir wie folgt:

$$\begin{array}{rcl} \xi(x_\tau) & := & x_{\xi(\tau)}, \\ \xi(f_\tau(t_1, \ldots, t_n)) & := & f_{\xi(\tau)}(\xi(t_1), \ldots, \xi(t_n)), \\ \xi(\texttt{let } x_\tau \texttt{ := } t \texttt{ in } r \texttt{ end}) & := & \texttt{let } x_{\xi(\tau)} \texttt{ := } \xi(t) \texttt{ in } \xi(r) \texttt{ end}. \end{array}$$

Für eine Signatur $\langle \Omega, \Sigma \rangle$ nennen wir eine Abbildung $\sigma : \mathcal{V}(\Omega) \to \mathcal{T}erm(\Omega, \Sigma)$ mit $\sigma(x_\tau) \neq x_\tau$ für nur endlich viele $x_\tau \in \mathcal{V}(\Omega)$ und $\sigma(x_\tau) \in \mathcal{T}erm_\tau(\Omega, \Sigma)$ für alle $x_\tau \in \mathcal{V}(\Omega)$ *Termsubstitution über* $\langle \Omega, \Sigma \rangle$. Die Menge aller Termsubstitutionen über $\langle \Omega, \Sigma \rangle$ bezeichnen wir mit $\mathcal{S}ubst(\Omega, \Sigma)$. Für eine Termsubstitution σ heißt dann die Menge

$$dom(\sigma) := \{x_\tau \in \mathcal{V}(\Omega) \mid \sigma(x_\tau) \neq x_\tau\}$$

Domain von σ und die Menge

$$rg(\sigma) := \{\sigma(x_\tau) \mid x_\tau \in dom(\sigma)\}$$

Range von σ. Gilt $dom(\sigma) = \{x_1, \ldots, x_n\}$, so notieren wir die Termsubstitution σ mit $\{x_1/\sigma(x_1), \ldots, x_n/\sigma(x_n)\}$. Die leere Termsubstitution $\{\}$ notieren wir ebenfalls mit ϵ. Für eine Menge von Termvariablen V bezeichnen wir mit $\mathcal{S}ubst_V(\Omega, \Sigma)$ die Menge aller Termsubstitutionen σ mit $dom(\sigma) = V$. Für eine Termsubstitution σ und eine Menge V definieren wir die auf V beschränkte Termsubstitution $\sigma|_V$ durch

$$\begin{array}{rcll} \sigma|_V(x) & := & \sigma(x) & , \text{ falls } x \in V, \\ \sigma|_V(x) & := & x & , \text{ sonst.} \end{array}$$

Die homomorphe Erweiterung einer Termsubstitution σ auf Terme definieren wir wie folgt:

$$\sigma(f_\tau(t_1, \ldots, t_n)) := f_\tau(\sigma(t_1), \ldots, \sigma(t_n))$$

$$\sigma(\texttt{let } x_\tau\texttt{:=}t \texttt{ in } r \texttt{ end}) := \begin{cases} \texttt{let } x_\tau\texttt{:=}\sigma(t) \texttt{ in } \sigma|_{dom(\sigma)\setminus\{x_\tau\}}(r) \texttt{ end}, \\ \qquad \text{falls } x_\tau \notin fv(rg(\sigma|_{fv(r)})), \\ \texttt{let } x'_\tau\texttt{:=}\sigma(t) \texttt{ in } \sigma|_{dom(\sigma)\setminus\{x'_\tau\}}(r[x_\tau/x'_\tau]) \texttt{ end}, \\ \qquad \text{sonst (mit } x'_\tau = Var_\tau(fv(r) \cup fv(rg(\sigma|_{fv(r)})))). \end{cases}$$

Ist $dom(\sigma) = \{x_1, \ldots, x_n\}$, so schreiben wir für $\sigma(t)$ auch $t[x_1/\sigma(x_1), \ldots, x_n/\sigma(x_n)]$.

Wir bezeichnen einen Term t als *Instanz* eines Terms r, falls eine Typsubstitution ξ und eine Termsubstitution σ existieren mit $t = \sigma(\xi(r))$. Wir notieren das Paar $\langle \xi, \sigma \rangle$ dann auch kurz mit σ_ξ und schreiben für $\sigma(\xi(r))$ auch kurz $\sigma_\xi(r)$.

Die Teilterme eines Terms adressieren wir mir Hilfe von Positionen.

[4] Die Typsubstitution ξ_t ist eindeutig bestimmt. Angenommen, es existieren zwei unterschiedliche $\subseteq_{dom}$-minimale Typsubstitutionen ξ und $\bar{\xi}$ mit $\xi(\tau'_i) = \tau_i$ und $\bar{\xi}(\tau'_i) = \tau_i$ für alle $i \in \{1, \ldots, n+1\}$. Da beide Typsubstitutionen $\subseteq_{dom}$-minimal sind, gilt dann $dom(\xi), dom(\bar{\xi}) \subseteq tv(\tau'_1) \cup \ldots \cup tv(\tau'_{n+1})$. Somit existiert ein $i \in \{1, \ldots, n+1\}$ und ein $\nu \in tv(\tau'_i)$ mit $\xi(\nu) \neq \bar{\xi}(\nu)$. Daraus folgt, dass $\xi(\tau'_i) \neq \bar{\xi}(\tau'_i)$ gilt. Da aber nach Voraussetzung $\xi(\tau'_i) = \tau_i$ und $\bar{\xi}(\tau'_i) = \tau_i$ gilt, ist dies ein Widerspruch zur Voraussetzung.

Definition 3.5 (Positionen)**.** Sei $\langle \Omega, \Sigma \rangle$ eine Signatur und $t \in \mathcal{Term}(\Omega, \Sigma)$. Die Menge $\mathcal{Pos}(t) \subseteq \mathbb{N}^*$ der Positionen von t ist die kleinste Menge mit

(1) $\epsilon \in \mathcal{Pos}(t)$,

(2) $i.\pi \in \mathcal{Pos}(t)$, falls $t = f(t_1, \ldots, t_n)$, $\pi \in \mathcal{Pos}(t_i)$ und $i \in \{1, \ldots, n\}$,

(3) $i.\pi \in \mathcal{Pos}(t)$, falls $t = \texttt{let}\ x\ \texttt{:=}\ t_1\ \texttt{in}\ t_2\ \texttt{end}$, $\pi \in \mathcal{Pos}(t_i)$ und $i \in \{1, 2\}$.

Für eine Position $\pi \in \mathcal{Pos}(t)$ ist der Teilterm $t|_\pi$ von t an Position π definiert durch

(1) $t|_\epsilon := t$,

(2) $f(t_1, \ldots, t_n)|_{i.\pi} := t_i|_\pi$,

(3) $\texttt{let}\ x\ \texttt{:=}\ t_1\ \texttt{in}\ t_2\ \texttt{end}|_{i.\pi} := t_i|_\pi$.

Für eine Position $\pi \in \mathcal{Pos}(t)$ mit $t|_\pi \in \mathcal{Term}_\tau(\Omega, \Sigma)$ und einen Term $s \in \mathcal{Term}_\tau(\Omega, \Sigma)$ ist die Teiltermersetzung $t[\pi \leftarrow s]$ definiert durch

(1) $t[\epsilon \leftarrow s] := s$,

(2) $f(t_1, \ldots, t_i, \ldots, t_n)[i.\pi \leftarrow s] := f(t_1, \ldots, t_i[\pi \leftarrow s], \ldots, t_n)$,

(3) $\texttt{let}\ x\ \texttt{:=}\ t_1\ \texttt{in}\ t_2\ \texttt{end}[1.\pi \leftarrow s] := \texttt{let}\ x\ \texttt{:=}\ t_1[\pi \leftarrow s]\ \texttt{in}\ t_2\ \texttt{end}$,

(4) $\texttt{let}\ x\ \texttt{:=}\ t_1\ \texttt{in}\ t_2\ \texttt{end}[2.\pi \leftarrow s] := \texttt{let}\ x\ \texttt{:=}\ t_1\ \texttt{in}\ t_2[\pi \leftarrow s]\ \texttt{end}$.

□

Auf Positionen definieren wir die folgende totale Ordnung:

$$\pi_1 >_{\mathcal{Pos}} \pi_2 \quad \text{gdw.} \quad \begin{array}{l}(\pi_1 \neq \epsilon \wedge \pi_2 = \epsilon) \vee \\ (\pi_1 = i_1.\pi_1 \wedge \pi_2 = i_2.\pi_2 \text{ mit } i_1 > i_2 \vee (i_1 = i_2 \wedge \pi_1' >_{\mathcal{Pos}} \pi_2')).\end{array}$$

Nachfolgend bezeichnen wir mit $t, t', t_1, t_2, \ldots, r, , r', r_1, r_2, \ldots$ immer Terme, mit $x, x', x_1, x_2, \ldots, y, y', y_1, y_2, \ldots$ Termvariablen, mit $\sigma, \sigma', \sigma_1, \sigma_2, \ldots$ Termsubstitutionen und mit $\pi, \pi', \pi_1, \pi_2, \ldots$ Positionen.

Beispiel 3.6. Sei Ω die Typsignatur aus Beispiel 3.2 und Σ die folgende Termsignatur:

$$\begin{array}{lcl}\{\texttt{0} & : & \rightarrow \texttt{nat}, \\ \texttt{succ} & : & \texttt{nat} \rightarrow \texttt{nat}, \\ \texttt{empty} & : & \rightarrow \texttt{list}[\texttt{@a}], \\ \texttt{add} & : & \texttt{@a} \times \texttt{list}[\texttt{@a}] \rightarrow \texttt{list}[\texttt{@a}]\}\ .\end{array}$$

Weiter sei $\texttt{x}_{\texttt{nat}}$ eine Termvariable. Die Ausdrücke

$$\begin{array}{rcl}t_1 & = & \texttt{add}(\texttt{x}, \texttt{add}(\texttt{x}, \texttt{empty}_{\texttt{list[nat]}})), \\ t_2 & = & \texttt{let x:=empty}_{\texttt{list[nat]}}\ \texttt{in add}(\texttt{0}, \texttt{x})\ \texttt{end}\end{array}$$

sind dann Terme über der Signatur $\langle \Omega, \Sigma \rangle$. Der Term t_1 ist `let`-frei und der Term t_2 ist nicht `let`-frei. Die Mengen der freien bzw. gebundenen Termvariablen der Terme sind wie folgt gegeben:

$$\begin{array}{rclcrcl}fv(t_1) & = & \{\texttt{x}\}, & & bv(t_1) & = & \emptyset, \\ fv(t_2) & = & \emptyset, & & bv(t_2) & = & \{\texttt{x}\}.\end{array}$$

Weiter ist die Abbildung

$$\sigma \quad = \quad \{\texttt{x/0}\}$$

eine Termsubstitutionen über $\langle \Omega, \Sigma \rangle$. Die zugehörige Domain und die zugehörige Range sind gegeben durch $dom(\sigma) = \{\mathtt{x}\}$ und $rg(\sigma) = \{\mathtt{0}\}$. Es gilt

$$\begin{array}{lcl} \sigma(t_1) & = & \mathtt{add}(\mathtt{0}, \mathtt{add}(\mathtt{0}, \mathtt{empty}_{\mathtt{list[nat]}})), \\ \sigma(t_2) & = & \mathtt{let\ x:=empty}_{\mathtt{list[nat]}}\ \mathtt{in\ add}(\mathtt{0}, \mathtt{x})\ \mathtt{end}. \end{array}$$

Die Menge der Positionen für t_1 ist gegeben durch

$$\mathcal{P}os(t_1) \quad = \quad \{\epsilon, \langle 1 \rangle, \langle 2 \rangle, \langle 2,1 \rangle, \langle 2,2 \rangle\}$$

und es gilt $\langle 2,2 \rangle >_{\mathcal{P}os} \langle 2,1 \rangle >_{\mathcal{P}os} \langle 2 \rangle >_{\mathcal{P}os} \langle 1 \rangle >_{\mathcal{P}os} \epsilon$ sowie

$$t_1[\langle 2,1 \rangle \leftarrow \mathtt{0}] = \mathtt{add}(\mathtt{x}, \mathtt{add}(\mathtt{0}, \mathtt{empty})).$$

□

3.3 Syntax von $\mathcal{L}$

Sowohl Typoperator- als auch Prozedurdefinitionen bauen auf vordefinierten Typoperatoren und Funktionssymbolen auf. Die Stelligkeiten dieser vordefinierten Typoperatoren bzw. Funktionssymbolen sind durch die Typsignatur

$$\Omega_0 \quad := \quad \{\mathtt{bool} : \mathtt{0}\}$$

und die Termsignatur

$$\Sigma_0 \quad := \quad \Sigma_0^{=} \ \cup \ \Sigma_0^{\mathrm{cons}} \ \cup \ \Sigma_0^{\mathrm{case}}$$

festgelegt, wobei gilt

$$\Sigma_0^{=} \quad := \quad \{\ \mathtt{=} : \mathtt{@r} \times \mathtt{@r} \rightarrow \mathtt{bool}\ \},$$

$$\Sigma_0^{\mathrm{cons}} \quad := \quad \{\ \mathtt{false} : \rightarrow \mathtt{bool}, \mathtt{true} : \rightarrow \mathtt{bool}\ \}$$

$$\Sigma_0^{\mathrm{case}} \quad := \quad \{\ \mathtt{case}^{\{\mathtt{false},\mathtt{true}\}} : \mathtt{bool} \times \mathtt{@r} \times \mathtt{@r} \rightarrow \mathtt{@r}\ \}$$

Der Typ `bool` repräsentiert die boolschen Wahrheitswerte. Wir bezeichnen `false` und `true` als die Konstruktoren von `bool`. Für $\mathtt{=}(t_1, t_2)$ schreiben wir auch $t_1 \mathtt{=} t_2$.

Aufbauend auf der Signatur $\langle \Omega_0, \Sigma_0 \rangle$ können wir nun Typoperator- und Prozedurdefinitionen formal definieren. Hierzu treffen wir noch folgende Vereinbarung: Wir gehen davon aus, dass $* \in \mathcal{S}$ gilt. Weiter gehen wir davon aus, dass durch keine Typoperator- und keine Prozedurdefinition $*$ als neues Funktionssymbol in ein Programm eingeführt wird. Vielmehr verwenden wir $*$ ausschließlich als Repräsentation der Unbestimmtheit innerhalb unvollständig-definierter Prozeduren.

Definition 3.7 (Typoperatordefinitionen)**.** Sei Ω eine Typsignatur mit $\Omega_0 \subseteq \Omega$ und Σ eine Termsignatur mit $\Sigma_0 \subseteq \Sigma$. Seien weiter $\nu_1, \ldots, \nu_l$ paarweise verschiedene Typvariablen, ζ ein Typoperator und $cons_1, \ldots, cons_n$ und $sel_{1,1}, \ldots, sel_{1,m_1}, \ldots, sel_{n,1}, \ldots, sel_{n,m_n}$ paarweise verschiedene Funktionssymbole. Ein Ausdruck D der Form

$$\begin{array}{l} \mathtt{structure}\ \zeta[\nu_1, \ldots, \nu_l] \ \Leftarrow \\ \quad cons_1(sel_{1,1} : \tau_{1,1}, \ldots, sel_{1,m_1} : \tau_{1,m_1}), \\ \quad \ldots \\ \quad cons_n(sel_{n,1} : \tau_{n,1}, \ldots, sel_{n,m_n} : \tau_{n,m_n}) \end{array}$$

heißt dann *Typoperatordefinition bezüglich* $\langle \Omega, \Sigma \rangle$ gdw. für alle $i = 1, \ldots, n$ und alle $j = 1, \ldots, m_i$ folgendes gilt:

(1) $\tau_{i,j} \in \mathcal{Typ}(\Omega)$.

(2) $tv(\tau_{i,j}) \subseteq \{\nu_1, \ldots, \nu_l\}$.

(3) $\tau_{i,j} \geq_{\mathcal{Typ}} \zeta[\tau'_1, \ldots, \tau'_l] \implies \langle \tau'_1, \ldots, \tau'_l \rangle = \langle \nu_1, \ldots, \nu_l \rangle$.

Weiterhin muss mindestens ein $i = 1, \ldots, n$ existieren, so dass für alle $j = 1, \ldots, m_i$ gilt:

$$\tau_{i,j} \text{ enthält keinen Teiltyp der Form } \zeta[\nu_1, \ldots, \nu_l].$$

Schließlich muss für den Ausdruck D noch $\Omega(D) \subseteq \Omega$ sowie $\Sigma(D) \subseteq \Sigma$ gelten, wobei $\Omega(D)$ die Typsignatur

$$\{\zeta : l\}$$

bezeichnet und $\Sigma(D)$ die Termsignatur

$$\Sigma^{\text{cons}}(D) \cup \Sigma^{\text{sel}}(D) \cup \Sigma^{\text{case}}(D) \cup \Sigma^{?}(D)$$

mit

$$\begin{aligned}
\Sigma^{\text{cons}}(D) &:= \{\ cons_1 : \tau_{1,1} \times \ldots \times \tau_{1,m_1} \to \zeta[\nu^*],\ ^5 \\
&\quad \ldots, \\
&\quad cons_n : \tau_{n,1} \times \ldots \times \tau_{n,m_n} \to \zeta[\nu^*]\ \}, \\
\Sigma^{\text{sel}}(D) &:= \{\ sel_{1,1} : \zeta[\nu^*] \to \tau_{1,1}, \\
&\quad \ldots, \\
&\quad sel_{n,m_n} : \zeta[\nu^*] \to \tau_{n,m_n}\ \}, \\
\Sigma^{\text{case}}(D) &:= \{\ \texttt{case}^C : \zeta[\nu^*] \times \underbrace{\nu' \times \ldots \times \nu'}_{|C|\text{-mal}} \to \nu' \mid \\
&\quad C \subseteq \{cons_1, \ldots, cons_n\} \text{ mit } |C| > 1\}^6, \\
\Sigma^{?}(D) &:= \{\ ?cons_1 : \zeta[\nu^*] \to \texttt{bool}, \\
&\quad \ldots, \\
&\quad ?cons_n : \zeta[\nu^*] \to \texttt{bool}\ \}.
\end{aligned}$$

$\Omega(D)$ und $\Sigma(D)$ heißen die durch D definierte Typ- und Termsignatur. Die Funktionssymbole $cons_1, \ldots, cons_n$ heißen die *Konstruktoren* von ζ und die Funktionssymbole $sel_{i,1}, \ldots, sel_{i,m_i}$ heißen die zu $cons_i$ gehörigen *Selektoren*. Die Funktionssymbole $?cons_1, \ldots, ?cons_n$ heißen *Strukturprädikate* von ζ. □

Durch die Bedingung (1) der Definition wird sichergestellt, dass alle $\tau_{i,j}$ Typen sind. Bedingung (2) gewährleistet, dass nur Typvariablen in der Definition der Konstruktoren und Selektoren verwendet werden dürfen, die auch im Kopf

$$\texttt{structure}\ \zeta[\nu_1, \ldots, \nu_l]$$

der Typoperatordefinition angegeben wurden. Die Bedingung (3) verhindert schließlich die Definition unsinniger Typoperatoren wie beispielsweise

$$\begin{aligned}
&\texttt{structure list[@a]} \Leftarrow \\
&\quad \texttt{empty, add(hd : @a, tl : list[nat])}.^7
\end{aligned}$$

[5] Mit ν^* bezeichnen wir die Liste der Variablen $\nu_1, \ldots, \nu_l$.

[6] Mit ν' bezeichnen wir eine neue Typvariable, die für alle $i = 1, \ldots, n$ ungleich ν_i ist.

Terme der Form $\texttt{?}cons(t)$ bezeichnen wir nachfolgend als *Strukturtests*, Terme der Form $t\texttt{=}cons_i(sel_{i,1}(t), \ldots, sel_{i,m_i}(t))$ als *Strukturgleichungen* und Terme der Form $\texttt{case}^C(\ldots)$ als `case`-*Terme*. `case`-Terme definieren Fallunterscheidungen. Einen `case`-Term wie

$$\texttt{case}^{\{cons_{i_1},\ldots,cons_{i_l}\}}(b, t_1, \ldots, t_l)$$

mit $i_1 < \ldots < i_l$ notieren wir mit

$$\texttt{case}(b, cons_{i_1} : t_1, \ldots, cons_{i_l} : t_l).$$

Weiter notieren wir `case`-Terme der Form $\texttt{case}(b, \texttt{true} : t_1, \texttt{false} : t_2)$ mit $\texttt{if}(b, t_1, t_2)$ und bezeichnen sie als `if`-*Terme*. Die Terme t_i eines `case`-Terms nennen wir Zweige des `case`-Terms und den Term b Bedingung des `case`-Terms. Ist nicht für jeden Konstruktor eines Typoperators ein Zweig definiert, so sprechen wir von einer unvollständigen Fallunterscheidung. Mit jedem Zweig t_i assoziieren wir weiterhin einen Term $\texttt{?}(cons_i, t_i)$ des Typs `bool`, der festlegt, wann der Zweig ausgewertet wird. Der Term $\texttt{?}(cons_i, t_i)$ ist wie folgt definiert:

$$\texttt{?}(cons_i, t_i) \quad := \quad \begin{cases} t_i & , \text{ falls } cons_i = \texttt{true}, \\ \texttt{if}(t_i, \texttt{false}, \texttt{true}) & , \text{ falls } cons_i = \texttt{false}, \\ \texttt{?}cons_i(t_i) & , \text{ sonst.} \end{cases}$$

Schließlich nennen wir einen Term `case`-*frei*, falls er keinen `case`-Term als Teilterm enthält.

Beispiel 3.8. Der Ausdruck $D_{\texttt{list}}$ aus Abbildung 3.1 ist eine Typoperatordefinition bzgl.

$$\langle \Omega_0 \cup \{\texttt{list} : 1\}, \Sigma_0 \cup \Sigma^{\text{cons}}(D_{\texttt{list}}) \cup \Sigma^{\text{sel}}(D_{\texttt{list}}) \cup \Sigma^{\text{case}}(D_{\texttt{list}}) \cup \Sigma^{\texttt{?}}(D_{\texttt{list}}) \rangle,$$

wobei gilt

$$\begin{aligned} \Sigma^{\text{cons}}(D_{\texttt{list}}) &= \{\, \texttt{empty} :\rightarrow \texttt{list}[\texttt{@a}], \\ &\quad\ \texttt{add} : \texttt{@a} \times \texttt{list}[\texttt{@a}] \rightarrow \texttt{list}[\texttt{@a}] \,\}, \\ \Sigma^{\text{sel}}(D_{\texttt{list}}) &= \{\, \texttt{hd} : \texttt{list}[\texttt{@a}] \rightarrow \texttt{@a}, \\ &\quad\ \texttt{tl} : \texttt{list}[\texttt{@a}] \rightarrow \texttt{list}[\texttt{@a}] \,\}, \\ \Sigma^{\text{case}}(D_{\texttt{list}}) &= \{\, \texttt{case} : \texttt{list}[\texttt{@a}] \times \texttt{@r} \times \texttt{@r} \rightarrow \texttt{@r} \,\}, \\ \Sigma^{\texttt{?}}(D_{\texttt{list}}) &= \{\, \texttt{?empty} : \texttt{list}[\texttt{@a}] \rightarrow \texttt{bool}, \\ &\quad\ \texttt{?add} : \texttt{list}[\texttt{@a}] \rightarrow \texttt{bool} \,\}. \end{aligned}$$

Die Funktionssymbole `empty` und `add` sind die Konstruktoren, `hd` und `tl` die Selektoren und `?empty` und `?add` die Strukturprädikate des Typoperators `list`. □

Kommen wir nun zu Prozedurdefinitionen:

Definition 3.9 (Prozedurdefinitionen)**.** Sei Ω eine Typsignatur mit $\Omega_0 \subseteq \Omega$ und Σ eine Termsignatur mit $\Sigma_0 \subseteq \Sigma$. Seien weiter $x^1_{\tau_1}, \ldots, x^n_{\tau_n}$ paarweise

[7] Solche Definitionen sind aus theoretischer Sicht unproblematisch. In der Praxis deuten sie jedoch auf Eingabefehler des Benutzers hin. Aus praktischer Sicht ist es daher sinnvoll diese Definitionen als syntaktisch inkorrekt zu definieren.

$$
\begin{array}{rcl}
D_{\texttt{nat}} & := & \texttt{structure nat} \Leftarrow \texttt{0}, \texttt{succ}(\texttt{pred} : \texttt{nat}) \\
D_{\texttt{list}} & := & \texttt{structure list}[\texttt{@a}] \Leftarrow \texttt{empty}, \texttt{add}(\texttt{hd} : \texttt{@a}, \texttt{tl} : \texttt{list}[\texttt{@a}])
\end{array}
$$

Abbildung 3.1: Typoperatordefinitionen für `nat` und `list`.

---

verschiedene Termvariablen aus $\mathcal{V}(\Omega)$ und f ein Funktionssymbol. Ein Ausdruck D der Form

$$\texttt{function}\ f(x^1 : \tau_1, \ldots, x^n : \tau_n) : \tau \ \Leftarrow\ R_f$$

heißt *Prozedurdefinition bezüglich* $\langle \Omega, \Sigma \rangle$ gdw. die folgenden Bedingungen erfüllt sind:

(1) $fv(R_f) \subseteq \{x^1_{\tau_1}, \ldots, x^n_{\tau_n}\}$.

(2) $\Sigma(D) \subseteq \Sigma$.

(3) $R_f \in \mathcal{T}erm_\tau(\Omega, \Sigma \cup \{* : \tau_1 \times \ldots \times \tau_n \to \tau\})$.

(4) $R_f \geq_{\mathcal{T}erm} f_{\tau'}(t_1, \ldots, t_n) \Longrightarrow \xi_{f_{\tau'}(t_1,\ldots,t_n)} = \epsilon$.

(5) $R_f \geq_{\mathcal{T}erm} *(t_1, \ldots, t_n) \Longrightarrow \langle t_1, \ldots, t_n \rangle = \langle x^1_{\tau_1}, \ldots, x^n_{\tau_n} \rangle$.

Hierbei bezeichnet $\Sigma(D)$ die durch D definierte Termsignatur

$$\{f : \tau_1 \times \ldots \times \tau_n \to \tau\}\,.$$

Die durch D definierte Typsignatur $\Omega(D)$ ist definiert als $\emptyset$. Das Funktionssymbol f heißt dann *Prozedur* und der Term R_f der *Prozedurrumpf* (oder kurz *Rumpf*) von f. □

Für eine Liste der Form $x_1 : \tau, \ldots, x_n : \tau$ schreiben wir häufig kurz $x_1, \ldots, x_n : \tau$. Weiter notieren wir eine Liste der Form $x_1 : \tau_1, \ldots, x_n : \tau_n$ durch $x^* : \tau^*$, wobei x^* die Liste $x_1, \ldots, x_n$ bezeichnet und τ^* die Liste $\tau_1, \ldots, \tau_n$. Wir nennen eine Prozedur f *unvollständig-definiert*, falls der Prozedurrumpf R_f einen Teilterm der Form $*(t_1, \ldots, t_n)$ enthält. Andernfalls nennen wir f *vollständig-definiert*. Da die Argumente der Teilterme $*(t_1, \ldots, t_n)$ durch Bedingung (5) der Definition 3.9 festgelegt sind, schreiben wir für $*(t_1, \ldots, t_n)$ auch kurz $*$. Wir nennen eine Prozedur f *rekursiv-definiert*, falls der Prozedurrumpf einen Teilterm der Form $f(t_1, \ldots, t_n)$ enthält. Schließlich bezeichnen wir für eine Prozedur f einen Term der Form $f(t_1, \ldots, t_n)$ als *Prozeduraufruf*.

Durch Bedingung (4) der Definition wird sichergestellt, dass in jedem rekursiven Aufruf von f im Prozedurrumpf R_f die Argumente die gleichen Typen haben wie Parameter $x^1_{\tau_1}, \ldots, x^n_{\tau_n}$. Außerdem wird durch die Bedingung gewährleistet, dass das Ergebnis der rekursiven Aufrufe mit dem Rückgabetyp τ der Prozedur übereinstimmt. Mit diesen Festlegungen verhindern wir unsinnige Prozedurdefinitionen wie beispielsweise

```
function stupid(l : list[@a]) : nat ⇐
  if(?empty(l),0, succ(stupid(add(0, empty)))).
```
[8]

[8] Solche Definitionen sind aus theoretischer Sicht unproblematisch. In der Praxis deuten sie jedoch auf Eingabefehler des Benutzers hin. Aus praktischer Sicht ist es daher sinnvoll diese Definitionen als syntaktisch inkorrekt zu definieren.

$$
\begin{array}{rcl}
D_{>} & := & \texttt{function}\ \texttt{>}(\texttt{n} : \texttt{nat}, \texttt{m} : \texttt{nat}) : \texttt{bool} \Leftarrow \\
& & \quad \texttt{if}(\texttt{?0}(\texttt{x}), \\
& & \quad\quad \texttt{false}, \\
& & \quad\quad \texttt{if}(\texttt{?0}(\texttt{m}), \\
& & \quad\quad\quad \texttt{true}, \\
& & \quad\quad\quad \texttt{pred}(\texttt{n})\texttt{>}\texttt{pred}(\texttt{m})))
\end{array}
$$

$$
\begin{array}{rcl}
D_{\texttt{length}} & := & \texttt{function}\ \texttt{length}(\texttt{l} : \texttt{list}[\texttt{@a}]) : \texttt{nat} \Leftarrow \\
& & \quad \texttt{if}(\texttt{?empty}(\texttt{l}), \\
& & \quad\quad \texttt{0} \\
& & \quad\quad \texttt{succ}(\texttt{length}(\texttt{tl}(\texttt{l}))))
\end{array}
$$

Abbildung 3.2: Prozedurdefinition für > und `length`.

Beispiel 3.10. Der Ausdruck $D_{>}$ aus Abbildung 3.2 ist eine Prozedurdefinition bzgl. $\langle \Omega_0 \cup \Omega(D_{\texttt{nat}}), \Sigma_0 \cup \Sigma(D_{\texttt{nat}}) \cup \{\texttt{>} : \texttt{nat} \times \texttt{nat} \to \texttt{bool}\}\rangle$, wobei $D_{\texttt{nat}}$ die entsprechende Typoperatordefinition aus Abbildung 3.1 bezeichnet. Die durch $D_{>}$ definierte Termsignatur lautet

$$\Sigma(D_{>}) = \{\texttt{>} : \texttt{nat} \times \texttt{nat} \to \texttt{bool}\}.$$

Der Ausdruck $D_{\texttt{length}}$ aus Abbildung 3.2 ist eine Prozedurdefinition bzgl.

$$\langle \Omega_0 \cup \Omega(D_{\texttt{nat}}) \cup \Omega(D_{\texttt{list}}), \Sigma_0 \cup \Sigma(D_{\texttt{nat}}) \cup \Sigma(D_{\texttt{list}}) \cup \{\texttt{length} : \texttt{list}[\texttt{@a}] \times \texttt{nat}\}\rangle,$$

wobei $D_{\texttt{nat}}$ und $D_{\texttt{list}}$ die entsprechenden Typoperatordefinitionen aus Abbildung 3.1 bezeichnen. □

Nachdem wir nun Typoperator- und Prozedurdefinitionen eingeführt haben, definiert wir schließlich Programme.[9]

Definition 3.11 (Programme)**.** Eine endliche Liste $P = \langle D_1, \ldots, D_n\rangle$ von Typoperator- und Prozedurdefinitionen heißt ein *$\mathcal{L}$-Programm* (oder kurz *Programm*) gdw. folgende Bedingungen erfüllt sind:

(1) Für alle $i \in \{1, \ldots, n\}$ ist D_i eine Typoperator- oder Prozedurdefinition bzgl. $\langle \Omega_0 \cup \Omega(D_1) \cup \ldots \cup \Omega(D_i), \Sigma_0 \cup \Sigma(D_1) \cup \ldots \cup \Sigma(D_i)\rangle$.

(2) Für alle $D \in P$ gilt $dom(\Omega(D)) \cap \Omega_0 = \emptyset$ und $dom(\Sigma(D)) \cap \Sigma_0 = \emptyset$.

(3) Für alle $D, D' \in P$ mit $D \neq D'$ gilt $dom(\Omega(D)) \cap dom(\Omega(D')) = \emptyset$ und $dom(\Sigma(D)) \cap dom(\Sigma(D')) = \emptyset$.

Die durch ein Programm $P = \langle D_1, \ldots, D_n\rangle$ eingeführten Typ- und Termsignaturen sind wie folgt definiert:

$$\Omega(P) := \Omega_0 \cup \Omega(D_1) \cup \ldots \cup \Omega(D_n),$$

$$\Sigma(P) := \Sigma_0 \cup \Sigma(D_1) \cup \ldots \cup \Sigma(D_n),$$

[9] Das in ✓`eriFun` implementierte Terminierungsverfahren [48] kann gegenseitig-rekursive Prozeduren nicht behandeln. Wir beschränken uns daher im Folgenden auf Programme ohne gegenseitig-rekursive Prozeduren. Die in der Arbeit vorgestellten Verfahren und Heuristiken sind daher auf Basis von Programmen ohne gegenseitig-rekursive Prozeduren definiert.

$$\begin{aligned}
\Sigma^{\text{cons}}(P) &:= \Sigma_0^{\text{cons}} \cup \Sigma^{\text{cons}}(D_{i_1}) \cup \ldots \cup \Sigma^{\text{cons}}(D_{i_m}),\\
\Sigma^{\text{sel}}(P) &:= \Sigma_0^{\text{sel}} \cup \Sigma^{\text{sel}}(D_{i_1}) \cup \ldots \cup \Sigma^{\text{sel}}(D_{i_m}),\\
\Sigma^{\text{case}}(P) &:= \Sigma_0^{\text{case}} \cup \Sigma^{\text{case}}(D_{i_1}) \cup \ldots \cup \Sigma^{\text{case}}(D_{i_m}),\\
\Sigma^{?}(P) &:= \Sigma_0^{?} \cup \Sigma^{?}(D_{i_1}) \cup \ldots \cup \Sigma^{?}(D_{i_m}),\\
\Sigma^{\text{proc}}(P) &:= \Sigma_0^{>} \cup \Sigma(D_{j_1}) \cup \ldots \cup \Sigma(D_{j_l}).
\end{aligned}$$

Hierbei bezeichnen $D_{i_1}, \ldots, D_{i_m}$ genau die Typoperatordefinitionen von P und $D_{j_1}, \ldots, D_{j_l}$ die Prozedurdefinitionen von P. Weiterhin wird mit $\Sigma^{\text{rec}}(P)$ die Termsignatur der rekursiv-definierten Prozeduren von P bezeichnet und mit $\Sigma^{\text{n-rec}}(P)$ die Termsignatur der nicht-rekursiv definierten Prozeduren von P. □

Beispiel 3.12. Die Liste $P = \langle D_{\texttt{nat}}, D_{>}, D_{\texttt{list}}, D_{\texttt{length}} \rangle$ mit den Typoperatordefinitionen aus Abbildung 3.1 und den Prozedurdefinitionen aus Abbildung 3.2 definiert ein Programm. □

Jedes Programm definiert eine Signatur:

$$\langle \Omega(P), \Sigma(P) \rangle.$$

Wir identifizieren im Folgenden ein Programm P mit dieser Signatur und schreiben beispielsweise statt $\mathcal{T}\!erm(\Omega(P), \Sigma(P))$ einfach $\mathcal{T}\!erm(P)$. Es ist zu beachten, dass durch Bedingung (1) der Definition (3.11) sichergestellt ist, dass funktionale Programme keine gegenseitig-rekursiven Typoperator- und Prozedurdefinitionen enthalten.

Eine Typoperatordefinition für einen Typoperator ζ führt eine neue freie algebraische Datenstruktur in ein Programm P ein. Die `let`-freien Terme

$$q \in \mathcal{T}\!erm_{\zeta[\tau^*]}(\Omega(P), \Sigma^{\text{cons}}(P)) \text{ mit } \mathit{fv}(q) = \emptyset$$

repräsentieren hierbei die Elemente dieser freien Datenstruktur. Wir bezeichnen diese Terme im Folgenden als *Konstruktorgrundterme von P*. Die Menge der Konstruktorgrundterme des Typs τ eines Programms P notieren wir dann mit $\mathcal{T}\!erm_{\tau}^{\text{cons}}(P)$ und die Menge aller Konstruktorgrundterme notieren wir mit $\mathcal{T}\!erm^{\text{cons}}(P)$. Für jeden monomorphen Typ τ gilt $\mathcal{T}\!erm_{\tau}^{\text{cons}}(P) \neq \emptyset$.

3.4 Semantik von $\mathcal{L}$

Wir definieren in diesem Abschnitt die Semantik von $\mathcal{L}$. Das heißt wir legen fest, wie den Termen t eines Programms P eine Deutung in Form von Konstruktorgrundtermen zugewiesen werden kann. Wie wir bereits in der Einleitung zu diesem Kapitel beschrieben haben, sind nicht notwendigerweise alle Funktionssymbole, die durch ein $\mathcal{L}$-Programm eingeführt werden vollständig-definiert. Die folgenden Funktionssymbole können in $\mathcal{L}$ unvollständig-definiert sein:

(1) **Prozeduren:** Durch die Verwendung von `*` im Rumpf R_f einer Prozedur f wird f als unvollständig-definiertes Funktionssymbol eingeführt. So wird

beispielsweise durch die Prozedurdefinition

```
function minimum(l : list[nat]) : nat ⇐
  if(?empty(l),
     *,
     if(?empty(tl(l)),
        hd(l),
        if(hd(l)>minimum(tl(l)),
           minimum(tl(l)),
           hd(l))))
```
(3.2)

das Ergebnis für `minimum(empty)` nicht festgelegt. Die Prozedur `minimum` ist somit unvollständig-definiert.

(2) **Selektoren:** Die Selektoren einer Typoperatordefinition mit mehr als einem Konstruktor sind unvollständig-definiert. So legt zwar beispielsweise die Typoperatordefinition

```
structure list[@a] ⇐
  empty, add(hd : @a, tl : list[@a])
```
(3.3)

die Bedeutung von Termen wie `hd(add(1, empty))` fest, für Term wie `hd(empty)` ist jedoch durch die Typoperatordefinition keine Deutung definiert.

(3) `case`-**Funktionen:** Die durch eine Typoperatordefinition eingeführten `case`-Funktionen sind nicht notwendigerweise vollständig-definiert. Die Typoperatordefinition

```
structure tree[@a] ⇐
  nil,
  atom(index : @a),
  node(left : tree[@a], right : tree[@a])
```
(3.4)

führt unter anderem die `case`-Funktion $\texttt{case}^{\{\texttt{nil},\texttt{atom}\}}$ ein. Für diese `case`-Funktion ist zwar das Ergebnis des Terms

$$\texttt{case}^{\{\texttt{nil},\texttt{atom}\}}(\texttt{nil}, 0, 1)$$

definiert, nicht jedoch das Ergebnis des Terms

$$\texttt{case}^{\{\texttt{nil},\texttt{atom}\}}(\texttt{node}(\texttt{nil},\texttt{nil}), 0, 1).$$

Unvollständig-definierten Funktionssymbolen kann nicht zwangsläufig eine eindeutige Semantik zugewiesen werden. Das führt dazu, dass nicht jedem Term eine eindeutige Interpretation zugewiesen werden kann. Die Definition unserer Semantik berücksichtigt diese Tatsache und interpretiert Typen und Terme bzgl. so genannter P-Interpretationen. Diese P-Interpretationen deuten alle Ausdrücke, deren Deutung durch die unvollständige Definition eines Programms P offen geblieben sind. Eine Aussage über ein Programm ist dann gültig, wenn sie für *jede* P-Interpretation und somit auch für jede Beispieltermfunktion gültig ist. Die Definition unserer Semantik erfolgt in zwei Schritten. Zunächst definieren wir im ersten Schritt eine Semantik für Programme ohne unvollständig-definierte Prozeduren (Definition 3.18). Im zweiten Schritt definieren wir dann auf Basis dieser Semantik mit Hilfe von P-Interpretationen die Semantik für Programme mit unvollständig-definierten Prozeduren (Definition 3.21).

Zur Definition der Semantik für Programme ohne unvollständig-definierte Prozeduren müssen wir festlegen, wie Terme der Form $sel(q)$ mit $sel \in \Sigma^{\text{sel}}(P)$ und

$\texttt{case}^C(q,\ldots)$ mit $\texttt{case}^C \in \Sigma^{\text{case}}(P)$ interpretiert werden sollen, falls durch die Typoperatordefinition keine entsprechenden Deutungen festlegt sind. Betrachten wir dazu erneut die Typoperatordefinition (3.4). Um dem Term

$$\texttt{index}_\tau(\texttt{nil}) \tag{3.5}$$

für jeden Typ τ eine Deutung zuweisen zu können, wird dem Selektor `index` für jeden Typ τ ein so genannter *Beispielterm* zugeordnet, durch den der Term gedeutet werden soll. Allgemein findet eine solche Zuordnung von Beispieltermen durch so genannte *Beispieltermfunktionen* statt.

Definition 3.13 (Beispieltermfunktion)**.** Sei P ein Programm. Eine Funktion Δ heißt *Beispieltermfunktion bzgl. P* gdw. gilt

(1) $\Delta : \mathcal{Typ}^{\text{mono}}(P) \times \mathcal{Term}^{\text{cons}}(P) \to \mathcal{Term}^{\text{cons}}(P)$ und

(2) $\Delta(\tau, t) \in \mathcal{Term}^{\text{cons}}_\tau(P)$ für alle $\tau \in \mathcal{Typ}^{\text{mono}}(P)$ und alle $t \in \mathcal{Term}^{\text{cons}}(P)$.

□

Um die Verwendung von Beispieltermfunktionen zu illustrieren, betrachten wir ein Beispiel.

Beispiel 3.14. Sei P gegeben durch $\langle D_{\texttt{nat}}, D\rangle$, wobei $D_{\texttt{nat}}$ die entsprechende Typoperatordefinition aus Abbildung 3.1 bezeichnet und D die in (3.4) angegebene Typoperatordefinition. Zur vollständigen Deutung des Selektors `index` definieren wir die folgende Beispieltermfunktion:

$$\begin{array}{rcl}
\Delta(\texttt{nat},\texttt{nil}) & := & \texttt{0} \\
\Delta(\texttt{nat},\texttt{atom}(q)) & := & \texttt{succ(0)} \\
\Delta(\texttt{nat},\texttt{cons}(q_1,q_2)) & := & \texttt{succ(succ(0))} \\
 & & \\
\Delta(\texttt{tree[bool]},q) & := & \texttt{nil} \\
\Delta(\texttt{tree[nat]},q) & := & \texttt{nil} \\
 & & \\
\Delta(\texttt{tree[tree[bool]]},q) & := & \texttt{nil} \\
\vdots & \vdots & \vdots
\end{array}$$

Mit Hilfe von Δ deuten wir dann einen Term der Form $\texttt{index}_\tau(cons(q^*))$ mit $cons \neq \texttt{atom}$ durch

$$\Delta(\tau, cons(q^*)).$$

Auf diese Weise interpretieren wir den Term $\texttt{index}_{\texttt{nat}}(\texttt{nil})$ als `0` und den Term $\texttt{index}_{\texttt{nat}}(\texttt{cons}(\texttt{nil},\texttt{nil}))$ als $\texttt{succ}(\texttt{succ}(\texttt{0}))$. □

Wir wollen noch kurz auf zwei wichtige Punkte der Definition 3.13 eingehen:

(1) *Warum übergeben wir einer Beispieltermfunktion Δ zur Auswahl des Beispielterms das Argument $cons(q^*)$ des Selektors?* Zur Beantwortung dieser Frage nehmen wir an, dass Beispieltermfunktionen wie folgt definiert seien:

   (a) $\Delta : \mathcal{Typ}^{\text{mono}}(P) \to \mathcal{Term}^{\text{cons}}(P)$ und

   (b) $\Delta(\tau) \in \mathcal{Term}^{\text{cons}}_\tau(P)$.

   Terme wie $\texttt{index}_{\texttt{nat}}(\texttt{nil})$ und $\texttt{index}_{\texttt{nat}}(\texttt{cons}(\texttt{nil},\texttt{nil}))$ würden durch solche Beispieltermfunktionen immer durch identische Terme interpretiert werden. Dies würde dazu führen, dass Gleichungen wie beispielsweise

$$\texttt{index}_{\texttt{nat}}(\texttt{nil})\texttt{=}\texttt{index}_{\texttt{nat}}(\texttt{cons}(\texttt{nil},\texttt{nil}))$$

gültig wären. Es ist jedoch weder die Gültigkeit noch die Ungültigkeit dieser Gleichung beabsichtigt. Wie wir durch Beispiel 3.14 gesehen haben, vermeiden wir durch Berücksichtigung des Arguments des Selektors die Gültigkeit solcher Gleichungen.

(2) *Warum sind Beispieltermfunktionen nur für monomorphe Typen definiert?* Der Grund ist, dass für polymorphe Typen nicht zwangsläufig ein Beispielterm in Form eines Konstruktorgrundterms existiert. So ist beispielsweise die Menge $\mathcal{Term}^{\text{cons}}_{\texttt{@a}}(P)$ für jedes Programm P leer. Wir können also mit Hilfe von Beispieltermfunktionen Termen wie $\texttt{index}_{\texttt{@a}}(\texttt{nil})$ nicht ohne weiteres eine Deutung zuweisen. Wir beschränken uns daher bei der Definition unserer Semantik zunächst auf Terme, die keine Typen mit Typvariablen enthalten. Solche Terme wollen wir nachfolgend als *monomorphe Terme* bezeichnen. Einen monomorphen Term t mit $fv(t) = \emptyset$ nennen wir dann *monomorphen Grundterm*.

Betrachten wir nun Terme der Form $\texttt{case}^C_\tau(cons(q^*), \ldots)$ mit $cons \notin C$. Auch solchen Termen wollen wir mit Hilfe von Beispieltermfunktionen eine Deutung zuweisen. Das bedeutet, dass wir für jede case-Funktion $\texttt{case}^C$ eine Beispieltermfunktion $\Delta_{\texttt{case}^C}$ definieren und wir deuten einen Term der Form $\texttt{case}^C_\tau(cons(q^*), \ldots)$ mit $cons \notin C$ durch $\Delta_{\texttt{case}^C}(\tau, cons(q^*))$.

Aufbauend auf Beispieltermfunktionen können wir nun einen Kalkül zur Auswertung monomorpher Grundterme definieren.

Definition 3.15 (Kalkül zur Auswertung monomorpher Grundterme)**.** Sei P ein Programm ohne unvollständig-definierte Prozeduren. Sei weiter

- $\Delta^{\text{sel}} = (\Delta_{sel})_{sel \in \Sigma^{\text{sel}}(P)}$ eine Familie von Beispieltermfunktionen und
- $\Delta^{\text{case}} = (\Delta_c)_{c \in \Sigma^{\text{case}}(P)}$ ebenfalls eine Familie von Beispieltermfunktionen.

Die Auswertung monomorpher Grundterme ist dann durch den folgenden Kalkül definiert:

(1) **Sprache:** Monomorphe Grundterme über P.

(2) **Inferenzregeln:** Im Folgenden bezeichnen $q, q_1, \ldots, q_n$ Konstruktorgrundterme, $cons_1, \ldots, cons_m$ die verschiedenen Konstruktoren eines Typoperators, $sel^{i,j}_\tau$ den j-ten Selektor eines Konstruktors $cons_i$ und R_f den Prozedurrumpf einer Prozedur f. Weiter bezeichnen i und i' verschiedene natürliche Zahlen.

$$\frac{q_1{=}q_2}{\texttt{true}}, \quad \text{falls } q_1 = q_2 \qquad \frac{q_1{=}q_2}{\texttt{false}}, \quad \text{falls } q_1 \neq q_2$$

$$\frac{?cons_i(cons_i(q_1, \ldots, q_n))}{\texttt{true}} \qquad \frac{?cons_{i'}(cons_i(q_1, \ldots, q_n))}{\texttt{false}}$$

$$\frac{\texttt{let } x{=}t \texttt{ in } r \texttt{ end}}{\texttt{let } x{=}t' \texttt{ in } r \texttt{ end}}, \quad \text{falls } t \to_{P,\Delta^{\text{sel}},\Delta^{\text{case}}} t'$$

$$\frac{\texttt{let } x{=}q \texttt{ in } r \texttt{ end}}{r[x/q]}$$

$$\frac{\texttt{case}^C_\tau(b, cons_1 : t_1, \ldots, cons_m : t_m)}{\texttt{case}^C_\tau(b', cons_1 : t_1, \ldots, cons_m : t_m)}, \quad \text{falls } b \to_{P,\Delta^{\text{sel}},\Delta^{\text{case}}} b'$$

$$\frac{\texttt{case}^C_\tau(cons_i(q_1, \ldots, q_n), \ldots, cons_i : t_i, \ldots)}{t_i}, \quad \text{falls } cons_i \in C$$

$$\frac{\texttt{case}^C_\tau(cons_i(q_1, \ldots, q_n), \ldots)}{\Delta_{\texttt{case}^C}(\tau, cons_i(q_1, \ldots, q_n))}, \quad \text{falls } cons_i \notin C$$

$$\frac{sel^{i,j}_\tau(cons_i(q_1, \ldots, q_j, \ldots, q_n))}{q_j}$$

$$\frac{sel^{i,j}_\tau(cons_{i'}(q_1, \ldots, q_j, \ldots, q_n))}{\Delta_{sel^{i,j}}(\tau, cons_{i'}(q_1, \ldots, q_j, \ldots, q_n))}$$

$$\frac{f(t_1, \ldots, t_i, \ldots, t_n)}{f(t_1, \ldots, t'_i, \ldots, t_n)}, \quad \text{falls } t_i \to_{P,\Delta^{\text{sel}},\Delta^{\text{case}}} t'_i \text{ und } f \notin \Sigma^{\text{case}}(P)$$

$$\frac{f(q_1, \ldots, q_n)}{\xi(\xi_{f(q_1,\ldots,q_n)}(R_f))[x_1/q_1, \ldots, x_n/q_n]},$$

falls $f \in \Sigma^{\text{proc}}(P)$ gilt und ξ eine Typsubstitution ist, so dass $\xi(\xi_{f(q_1,\ldots,q_n)}(R_f))$ ein monomorpher Term ist.[10]

(3) **Deduktion:** Wir schreiben $t \to_{P,\Delta^{\text{sel}},\Delta^{\text{case}}} t'$, falls eine Ableitung der Form

$$\frac{t}{t'}, \quad \text{falls } B(t)$$

existiert und die Bedingung $B(t)$ erfüllt ist. Die transitive-reflexive Hülle der Relation $\to_{P,\Delta^{\text{sel}},\Delta^{\text{case}}}$ notieren wir mit $\to^*_{P,\Delta^{\text{sel}},\Delta^{\text{case}}}$. Gilt $t \to^*_{P,\Delta^{\text{sel}},\Delta^{\text{case}}} q$ und existiert kein q' mit $q \to_{P,\Delta^{\text{sel}},\Delta^{\text{case}}} q'$, so schreiben wir $t \to^!_{P,\Delta^{\text{sel}},\Delta^{\text{case}}} q$. □

Wie man mit Induktion nachweisen kann, ist die Relation $\to_{P,\Delta^{\text{sel}},\Delta^{\text{case}}}$ konfluent. Es existiert daher für jeden Term t höchstens ein Term q mit $t \to^!_{P,\Delta^{\text{sel}},\Delta^{\text{case}}} q$. Um mit dem Kalkül Terme interpretieren zu können, die sowohl Typ- als auch Termvariablen enthalten, führen wir *Typ- und Termvariablenbelegungen* ein.

[10] Die Typsubstitution ξ ist notwendig, da nicht jeder instantiierte Prozedurrumpf $\xi_{f(q_1,\ldots,q_n)}(R_f)$ ein monomorpher Term ist. Beispielsweise ist für die Prozedurdefinition `function f(x : nat) : bool ⇐ empty@a=empty@a` und den Prozeduraufruf `f(0)` der instantiierte Prozedurrumpf $\texttt{empty}_{@\texttt{a}}{=}\texttt{empty}_{@\texttt{a}}$ offensichtlich kein monomorpher Term. Die Typsubstitution ξ stellt also sicher, dass der durch einen Prozeduraufruf eingeführte Term wieder ein monomorpher Term ist. Die Typsubstitution ξ hat keinen Einfluss auf die weitere Auswertung.

Definition 3.16 (Typvariablenbelegung)**.** Sei P eine Programm. Eine Funktion

$$\alpha : \mathcal{V}_{\mathcal{T}yp} \to \mathcal{T}yp^{\text{mono}}(P)$$

heißt *Typvariablenbelegung bzgl.* P. Die homomorphe Erweiterung von α auf Typen und Terme ist wie folgt definiert:

(1) $\alpha(\zeta[\tau_1, \ldots, \tau_n]) := \zeta[\alpha(\tau_1), \ldots, \alpha(\tau_n)]$.

(2) $\alpha(x_\tau) := x_{\alpha(\tau)}$.

(3) $\alpha(\texttt{let}\ x_\tau\texttt{:=}r\ \texttt{in}\ t\ \texttt{end}) := \texttt{let}\ x_{\alpha(\tau)}\texttt{:=}\alpha(r)\ \texttt{in}\ \alpha(t)\ \texttt{end}$.

(4) $\alpha(f_\tau(t_1, \ldots, t_n) := f_{\alpha(\tau)}(\alpha(t_1), \ldots, \alpha(t_n))$.

□

Definition 3.17 (Termvariablenbelegung)**.** Sei P ein Programm. Eine Funktion

$$a : \mathcal{V}^{\text{mono}}(P(\Omega)) \to \mathcal{T}erm^{\text{cons}}(P)$$

mit $a(x_\tau) \in \mathcal{T}erm^{\text{cons}}_\tau(P)$ heißt *Termvariablenbelegung bzgl.* P. Die homomorphe Erweiterung von a auf Terme ist wie folgt definiert:

(1) $a(\texttt{let}\ x_\tau\texttt{:=}r\ \texttt{in}\ t\ \texttt{end}) := \texttt{let}\ x_\tau\texttt{:=}a(r)\ \texttt{in}\ a(t)\ \texttt{end}$.

(2) $a(f_\tau(t_1, \ldots, t_n) := f_\tau(a(t_1), \ldots, a(t_n))$.

□

Wir bezeichnen im Folgenden mit $\alpha, \alpha', \ldots$ immer Typvariablenbelegungen und mit $a, a', \ldots$ Termvariablenbelegungen. Da durch die Bezeichnung immer klar ist, ob es sich um eine Typ- oder Termvariablenbelegung handelt, sprechen wir auch kurz von Variablenbelegungen. Auf Basis von Variablenbelegungen können wir nun die Semantik von Programmen ohne unvollständig-definierte Prozeduren definieren.

Definition 3.18 ($[\![\cdot]\!]^{\alpha,a}_{P,\Delta^{\text{sel}},\Delta^{\text{case}}}$)**.** Sei P ein Programm ohne unvollständig-definierte Prozeduren und α und a Variablenbelegungen bzgl. P. Sei weiter

- $\Delta^{\text{sel}} = (\Delta_{sel})_{sel \in \Sigma^{\text{sel}}(P)}$ eine Familie von Beispieltermfunktionen und
- $\Delta^{\text{case}} = (\Delta_c)_{c \in \Sigma^{\text{case}}(P)}$ ebenfalls eine Familie von Beispieltermfunktionen.

Die Deutung $[\![t]\!]^{\alpha,a}_{P,\Delta^{\text{sel}},\Delta^{\text{case}}}$ eines Terms $t \in \mathcal{T}erm(P)$ bzgl. P, Δ^{sel}, Δ^{case}, α und a ist definiert durch

$$[\![t]\!]^{\alpha,a}_{P,\Delta^{\text{sel}},\Delta^{\text{case}}} := \begin{cases} q & \text{, falls } a(\alpha(t)) \to^{!}_{P,\Delta^{\text{sel}},\Delta^{\text{case}}} q \\ \bot & \text{, sonst.}^{11} \end{cases}$$

Wir bezeichnen einen Term $t \in \mathcal{T}erm(P)$ als *P-definiert* gdw. für alle Variablenbelegungen α und a und alle Familien von Beispieltermfunktionen $\Delta^{\text{sel}} = (\Delta_s)_{s \in \Sigma^{\text{sel}}(P)}$ und $\Delta^{\text{case}} = (\Delta_c)_{c \in \Sigma^{\text{case}}(P)}$ gilt

$$[\![t]\!]^{\alpha,a}_{P,\Delta^{\text{sel}},\Delta^{\text{case}}} \neq \bot.$$

□

Wir kommen nun zur Definition der Semantik beliebiger Programme. Hierzu verwenden wir faire Vervollständigungen von Programmen [59]. Die Grundidee der fairen Vervollständigungen besteht darin, alle Vorkommen von $*$ in den Funktionsrümpfen der unvollständig-definierten Prozeduren durch beliebige Terme zu ersetzen, wobei das Terminierungsverhalten des Programms durch diese Ersetzungen nicht verändert werden darf. Die formale Definition der fairen Vervollständigungen sieht wie folgt aus:

Definition 3.19 (Faire Vervollständigung)**.** Sei P ein Programm. Eine *faire Vervollständigung* von P ist jedes Programm $\widehat{P}$ ohne unvollständig-definierte Funktionsprozeduren, dass die folgenden Bedingungen erfüllt:

(1) Jede Typoperatordefinition aus P und jede vollständig-definierte Prozedurdefinition aus P ist in $\widehat{P}$ enthalten.

(2) Für jede unvollständig-definierte Prozedurdefinition

$$\texttt{function}\ f(x^* : \tau^*) : \tau \ \Leftarrow\ R_f$$

in P existiert eine vollständig-definierte Prozedurdefinition

$$\texttt{function}\ f(x^* : \tau^*) : \tau \ \Leftarrow\ R'_f$$

in $\widehat{P}$, so dass gilt:

(a) $R_f = R'_f[\pi_1 \leftarrow *, \ldots, \pi_k \leftarrow *]$ mit $\{\pi_1, \ldots, \pi_k\} = \mathcal{P}os^*(R_f)$ und

(b) $\forall \pi \in \mathcal{P}os^*(R_f)$ ist $R'_f|_\pi$ $\widehat{P}$-definiert.

Hierbei ist $\mathcal{P}os^*(R_f)$ wie folgt definiert:

$$\mathcal{P}os^*(R_f) := \{\pi \in \mathcal{P}os(R_f) \mid R_f|_\pi = *\}.$$

□

Auf Basis der fairen Vervollständigungen definieren wir *P-Interpretationen*. Diese P-Interpretationen deuten, wie bereits gesagt, alle Ausdrücke, deren Deutung durch die unvollständige Definition eines Programms P offen geblieben sind.

Definition 3.20 (P-Interpretation)**.** Sei P ein Programm. Eine *P-Interpretation* ist eine Tupel der Form $\langle \widehat{P}, \Delta^{\text{sel}}, \Delta^{\text{case}} \rangle$, wobei gilt

(1) $\widehat{P}$ ist eine faire Vervollständigung von P,

(2) $\Delta^{\text{sel}} = (\Delta_{sel})_{sel \in \Sigma^{\text{sel}}(\widehat{P})}$ ist eine Familie von Beispieltermfunktionen bzgl. $\widehat{P}$,

(3) $\Delta^{\text{case}} = (\Delta_c)_{c \in \Sigma^{\text{case}}(\widehat{P})}$ ist eine Familie von Beispieltermfunktionen bzgl. $\widehat{P}$.

Die Komponenten einer P-Interpretation I bezeichnen wir im Folgenden mit $\widehat{P}_I$, Δ_I^{sel} und Δ_I^{case}. □

Wir bezeichnen Variablenbelegungen α und a als *I-Variablenbelegungen* falls die Variablenbelegungen bzgl. der fairen Vervollständigung $\widehat{P}_I$ definiert sind. Die Semantik eines beliebigen Programms P bzgl. einer P-Interpretation ist dann wie folgt definiert:

[11] Mit dem Funktionsergebnis $\perp$ drücken wir aus, dass die Funktion an dieser Stelle undefiniert ist.

Definition 3.21 ($[\![\cdot]\!]^{\alpha}_{P,I}$ und $[\![\cdot]\!]^{\alpha,a}_{P,I}$)**.** Sei P ein Programm und I eine P-Interpretation. Seien weiter α und a I-Variablenbelegung. Die Semantik $[\![\tau]\!]^{\alpha}_{P,I}$ eine Typs $\tau \in \mathcal{Typ}(P)$ bzgl. I und α ist definiert durch

$$[\![\tau]\!]^{\alpha}_{P,I} := \mathcal{Term}^{\mathrm{cons}}_{\alpha(\tau)}(\widehat{P}_I).$$

Die Semantik $[\![t]\!]^{\alpha,a}_{P,I}$ eines Terms $t \in \mathcal{Term}(P)$ bzgl. I, α und a ist definiert durch

$$[\![t]\!]^{\alpha,a}_{P,I} := [\![t]\!]^{\alpha,a}_{\widehat{P}_I,\Delta^{\mathrm{sel}}_I,\Delta^{\mathrm{case}}_I}.$$

□

Da wir an Verifikation interessiert sind, wollen wir auch Aussagen über Programme formulieren. Programmaussagen formulieren wir hierbei mittels boolscher Terme b, wobei die Termvariablen in b als universell quantifiziert angesehen werden. Bevor wir die Semantik solcher Programmaussagen definieren, betrachten wir dazu ein Beispiel:

Beispiel 3.22. Sei P' das Programm, das neben dem Typ `bool` nur aus der Prozedurdefinition

$$\texttt{function f(x : @a) : @a} \Leftarrow \texttt{*}$$

besteht. Dann ist

$$\texttt{if}(\neg \texttt{x}_{\texttt{@a}}\texttt{=f(x}_{\texttt{@a}}\texttt{)}, \texttt{if}(\neg \texttt{f(x}_{\texttt{@a}}\texttt{)=f(f(x}_{\texttt{@a}}\texttt{))}, \texttt{x}_{\texttt{@a}}\texttt{=f(f(x}_{\texttt{@a}}\texttt{))}, \texttt{false}), \texttt{false}) \tag{3.6}$$

eine Programmaussage über P'. Man kann jetzt argumentieren, dass diese Aussage für P' gültig ist, da in P' der Typ `bool` die einzig mögliche Ersetzung für `@a` ist und für diese Ersetzung die Aussage (3.6) gültig ist. Die allgemeine Argumentationsweise für die Gültigkeit von Aussagen über ein Programm P wäre demnach, dass eine Programmaussage Φ gilt, falls jede Instanz von Φ, in der Typvariable durch die monomorphen Typen von P und Termvariable durch Konstruktorgrundterme dieser monomorphen Typen ersetzt werden, gültig wäre:

> Für alle P-Interpretationen I und alle P-Variablenbelegungen α und a gilt
>
> $$[\![b]\!]^{\alpha,a}_{P,I} = \texttt{true}.$$

Allerdings ist diese Definition der Gültigkeit für die Verifikation von Programmen ungeeignet. Das Kernproblem ist hier, das Gültigkeit nicht monoton ist. Anders gesagt, wahre Programmaussagen können mit Erweiterung eines Programms falsch werden. Erweitert man beispielsweise das Programm P' um die Typoperatordefinition $D_{\texttt{nat}}$ (Abbildung 3.1), so gilt (3.6) offensichtlich nicht mehr. Da Programme jedoch inkrementell entwickelt werden, hat die Nicht-Monotonie zur Folge, dass alle bereits berechneten Beweise nach einer Programmerweiterung erneut auf ihre Gültigkeit hin überprüft werden müssen. Da dies offensichtlich für den praktischen Einsatz unsinnig ist, verwerfen wir diese Definition der Gültigkeit. □

Um das Problem der Nicht-Monotonie zu vermeiden, muss die Gültigkeit von Aussagen Φ über ein Programm P erhalten bleiben, wenn die Typvariablen in Φ durch beliebige monomorphe Typen (also insbesondere auch solche, die nicht mit den Typoperatordefinitionen von P gebildet werden können) ersetzt werden. Dies führt zu folgender Definition der Gültigkeit von Programmaussagen:

Definition 3.23 (Gültigkeit von Programmaussagen)**.** Wir bezeichnen einen boolschen Term als *P-gültig* oder kurz als *gültig* (in Zeichen $b \equiv_P \texttt{true}$), falls für alle P-Interpretationen I und alle I-Variablenbelegungen α und a gilt

$$[\![b]\!]^{\alpha,a}_{P,I} = \texttt{true}.$$

□

Durch die Verwendung von I-Variablenbelegungen in Definition 3.23 wird die Monotonie der Gültigkeit von boolschen Termen sichergestellt:

Lemma 3.24 (Monotonie der Gültigkeit)**.** Sei P ein Programm und $b \in \mathcal{T}erm(P)$ ein P-gültiger boolscher Term. Für jede Programmerweiterung P' von P ist dann b auch P'-gültig. □

Beweis. Sei I' eine beliebige aber feste P'-Interpretation und α' und a' beliebige aber feste I'-Variablenbelegungen. Da P' eine Programmerweiterung von P ist, ist I' auch eine P-Interpretation. Daraus folgt, dass $[\![b]\!]^{\alpha',a'}_{P,I'} = \texttt{true}$ gilt und damit auch $[\![b]\!]^{\alpha',a'}_{P',I'} = \texttt{true}$. Da I', α' und a' beliebig gewählt waren, folgt, dass b P'-gültig ist. □

Für die Deutung eines Terms sind lediglich die Deutungen der Typ- und Termvariablen relevant, die auch im Term enthalten sind. Wir schreiben daher häufig

$$[\![t]\!]^{[\nu_1:\alpha(\nu_1),\ldots,\nu_n:\alpha(\nu_1)][x_1:a(x_1),\ldots,x_m:a(x_m)]}_{P,I}$$

statt $[\![t]\!]^{\alpha,a}_{P,I}$, wobei $\{\nu_1,\ldots,\nu_n\} = tv(t)$ und $\{x_1,\ldots,x_m\} = fv(t)$ gilt.

Damit haben wir die Semantik der Sprache $\mathcal{L}$ vollständig definiert. Kommen wir nun zur Terminierung von $\mathcal{L}$-Programmen.

Definition 3.25 (Terminierung)**.** Sei P ein Programm. Eine Prozedur $f \in \Sigma^{\text{proc}}(P)$ mit $f : \tau_1 \times \ldots \times \tau_n \to \tau$ terminiert in P gdw. für alle P-Interpretationen I, alle I-Variablenbelegungen α und a und alle $q_1 \in [\![\tau_1]\!]^{\alpha}_{P,I}, \ldots, q_n \in [\![\tau_n]\!]^{\alpha}_{P,I}$ gilt

$$[\![f(x_1,\ldots,x_n)]\!]^{\alpha,[x_1:q_1,\ldots,x_n:q_n]}_{P,I} \neq \bot.$$

Eine Programm P terminiert gdw. jedes Funktionssymbol in P terminiert. □

3.5 Eigenschaften von Programmen

Nachdem wir nun die Syntax und Semantik der Programmiersprache von ✓eriFun definiert haben, wollen wir in diesem Abschnitt die wichtigsten Eigenschaften der Funktionssymbole eines Programms P angeben. Die Gültigkeit dieser Eigenschaften folgt direkt aus den Definitionen des Abschnitts 3.4. Wir verzichten daher auf die Angabe der entsprechenden Beweise. Die Eigenschaften sind insbesondere für das Verständnis des im zweiten Teil der Arbeit definierten Auswertungskalküls wichtig. Weiter definieren wir in diesem Abschnitt für die Funktionssymbole eines Programms die Begriffe assoziativ, kommutativ und transitiv.

Lemma 3.26 (Eigenschaften von Konstruktoren und Selektoren)**.** Sei P ein terminierendes Programm und $D \in P$ eine Typoperatordefinition der Form

$$\begin{aligned}&\texttt{structure}\ \zeta[\nu^*] \Leftarrow\\ &\quad cons_1(sel_{1,1} : \tau_{1,1}, \ldots, sel_{1,m_1} : \tau_{1,m_1}),\\ &\quad \ldots\\ &\quad cons_n(sel_{n,1} : \tau_{n,1}, \ldots, sel_{n,m_n} : \tau_{n,m_n}).\end{aligned}$$

Weiter seien $i, j \in \{1,\ldots,n\}$ mit $i \neq j$ und $k \in \{1,\ldots,m_i\}$ sowie

- $\mathtt{x}^* = \mathtt{x}_1, \ldots, \mathtt{x}_{m_i}$,
- $\bar{\mathtt{x}}^* = \bar{\mathtt{x}}_1, \ldots, \bar{\mathtt{x}}_{m_i}$,
- $\mathtt{y}^* = \mathtt{y}_1, \cdots, \mathtt{y}_{m_j}$,

- $A = AND(\mathtt{x}_1\mathtt{=}\bar{\mathtt{x}}_1, \ldots, \mathtt{x}_{m_i}\mathtt{=}\bar{\mathtt{x}}_{m_i})$.

Für die Konstruktoren und Selektoren von ζ gelten dann die folgenden Aussagen:

(1) $\mathtt{if}(cons_i(\mathtt{x}^*)\mathtt{=}cons_i(\bar{\mathtt{x}}^*), A, \neg A) \equiv_P \mathtt{true}$.

(2) $\neg(cons_i(\mathtt{x}^*)\mathtt{=}cons_j(\mathtt{y}^*)) \equiv_P \mathtt{true}$.

(3) $sel_{i,k}(cons_i(\mathtt{x}^*))\mathtt{=}\mathtt{x}^k \equiv_P \mathtt{true}$.

(4) $\mathtt{?}cons_i(cons_i(\mathtt{x}^*)) \equiv_P \mathtt{true}$.

(5) $\neg\mathtt{?}cons_j(cons_i(\mathtt{x}^*)) \equiv_P \mathtt{true}$.

□

Aussage (1) von Lemma 3.26 beschreibt die Injektivität von Konstruktoren. Das bedeutet, Konstruktoren repräsentieren injektive Funktionen. Aussage (2) von Lemma 3.26 beschreibt die Eindeutigkeit von Konstruktoren. Unter Eindeutigkeit verstehen wir hierbei die Tatsache, dass verschiedene Konstruktoren unterschiedliche Werte als Ergebnis liefern. Aussagen (3), (4) und (5) beschreiben schließlich die Semantik von Selektoren und Strukturprädikaten.

Lemma 3.27 (Eigenschaften von Strukturprädikaten)**.** Sei P ein terminierendes Programm und $D \in P$ eine Typoperatordefinition der Form

$$\begin{array}{l}\mathtt{structure}\ \zeta[\nu^*] \Leftarrow \\ \quad cons_1(sel_{1,1} : \tau_{1,1}, \ldots, sel_{1,m_1} : \tau_{1,m_1}), \\ \quad \ldots \\ \quad cons_n(sel_{n,1} : \tau_{n,1}, \ldots, sel_{n,m_n} : \tau_{n,m_n}).\end{array}$$

Weiter seien $i, j \in \{1, \ldots, n\}$ mit $i \neq j$. Für die Strukturprädikate von ζ gelten dann die folgenden Aussagen:

(1) $OR(\mathtt{?}cons_1(\mathtt{x}), \ldots, \mathtt{?}cons_n(\mathtt{x})) \equiv_P \mathtt{true}$.

(2) $\mathtt{if}(\mathtt{?}cons_i(\mathtt{x}), \neg\mathtt{?}cons_j(\mathtt{x}), \mathtt{true}) \equiv_P \mathtt{true}$.

(3) $\mathtt{if}(\mathtt{?}cons_i(\mathtt{x}), \mathtt{x}\mathtt{=}cons_i(sel_{i,1}(\mathtt{x}), \ldots, sel_{i,m_i}(\mathtt{x})), \mathtt{true}) \equiv_P \mathtt{true}$.

□

Aussagen (1) und (2) von Lemma 3.27 stellen sicher, dass für jeden $\zeta[\nu^*]$-Wert genau einer der Strukturtests gültig ist. Wir bezeichnen diese Eigenschaft auch als Vollständigkeit und Eindeutigkeit der Strukturtests.

Lemma 3.28 (Eigenschaft der Gleichheit)**.** Sei P ein terminierendes Programm. Für die Gleichheit = gelten dann die folgenden Aussagen:

(1) $\mathtt{x}\mathtt{=}\mathtt{x} \equiv_P \mathtt{true}$.

(2) $\mathtt{if}(\mathtt{x}\mathtt{=}\mathtt{y}, \mathtt{y}\mathtt{=}\mathtt{x}, \neg(\mathtt{y}\mathtt{=}\mathtt{x})) \equiv_\mathtt{P} \mathtt{true}$.

□

Aussage (2) von Lemma 3.28 beschreibt die Symmetrie der Gleichheit. Das bedeutet, dass wir die linke und rechte Seite einer Gleichung vertauschen können, ohne die Bedeutung der Gleichung zu verändern.

Lemma 3.29 (Eigenschaft von `case`)**.** Sei P ein terminierendes Programm und $D \in P$ eine Typoperatordefinition der Form

$$\begin{array}{l}\texttt{structure}\ \zeta[\nu^*] \Leftarrow \\ \quad cons_1(sel_{1,1} : \tau_{1,1}, \ldots, sel_{1,m_1} : \tau_{1,m_1}), \\ \quad \ldots \\ \quad cons_n(sel_{n,1} : \tau_{n,1}, \ldots, sel_{n,m_n} : \tau_{n,m_n}).\end{array}$$

Weiter sei $i \in \{1, \ldots, n\}$. Es gilt dann die folgende Aussage:

$$\texttt{case}(cons_i(\texttt{x}_1, \ldots, \texttt{x}_{m_i}), \ldots, cons_i : \texttt{y}, \ldots)\texttt{=y} \equiv_P \texttt{true}.$$

□

Für eine vollständig definierte Prozedur f mit Prozedurrumpf R_f gilt die folgende Aussage:

$$f(x^*)\texttt{=}R_f \equiv_P \texttt{true}. \tag{3.7}$$

Für unvollständig definierte Prozeduren ist der Prozedurrumpf jedoch kein Term über P und wir können daher auch nicht ohne weiteres eine Aussage der Form (3.7) für unvollständig definierte Prozeduren formulieren. Vielmehr ist es notwendig, jedes Vorkommen von $*$ im Prozedurrumpf durch einen entsprechenden Term über P zu ersetzen:

$$R_f^* := R_f[\pi_1 \leftarrow f(x^*), \ldots, \pi_n \leftarrow f(x^*)] \quad \text{mit } \{\pi_1, \ldots, \pi_n\} = \mathcal{P}os^*(R_f).$$

Es folgt dann dass folgende Lemma

Lemma 3.30 (Eigenschaft von Prozeduren)**.** Sei P ein Programm und $D \in P$ eine Prozedurdefinition der Form

$$\texttt{function}\ f(x_1 : \tau_1, \ldots, x_n : \tau_n) : \tau \Leftarrow R_f.$$

Für die Prozedur f gilt dann die folgende Aussage:

$$f(x^*)\texttt{=}R_f^* \equiv_P \texttt{true}.$$

□

Kommen wir nun zur Definition der Kommutativität, Assoziativität und Transitivität.

Definition 3.31 (Kommutativität, Assoziativität und Transitivität)**.** Sei P ein terminierendes Programm. Seien weiter $f, g \in \Sigma(P)$ Funktionssymbole mit

$$\begin{array}{rcl}\Sigma(P)(f) & = & \tau \times \tau \to \tau, \\ \Sigma(P)(g) & = & \tau \times \tau \to \texttt{bool}.\end{array}$$

Wir nennen

- f *kommutativ* bzgl. P (oder kurz kommutativ), falls gilt

$$f(\texttt{x}, \texttt{y})\texttt{=}f(\texttt{y}, \texttt{x}) \equiv_P \texttt{true}.$$

- f *assoziativ* bzgl. P (oder kurz assoziativ), falls gilt

$$f(\texttt{x}, f(\texttt{y}, \texttt{z}))\texttt{=}f(f(\texttt{x}, \texttt{y}), \texttt{z}) \equiv_P \texttt{true}.$$

- g *transitiv* bzgl. P (oder kurz transitiv), falls gilt

$$\texttt{if}(g(\texttt{x}, \texttt{y}), \texttt{if}(g(\texttt{y}, \texttt{z}), g(\texttt{x}, \texttt{z}), \texttt{true}), \texttt{true}) \equiv_P \texttt{true}.$$

□

```
D_tree = structure tree[@a] ⇐
           nil,
           leaf(value : @a),
           node(left : tree, right : tree)

D_elem = function elem(v : @a, t : tree[@a]) : bool ⇐
           if(?nil(t),
              false,
              if(?leaf(t),
                 value(t) = v,
                 if(elem(v, left(t)),
                    true,
                    elem(v, right(t)))))
```

Abbildung 3.3: Typoperator- und Prozedurdefinition von `tree` und `elem`.

3.6 Unvollständige Fallunterscheidungen

Wir gehen in diesem Abschnitt auf einige Fragen bezüglich unvollständiger Fallunterscheidungen ein, die bisher offen geblieben sind. Die hierbei betrachteten Beispiele sind auf Basis der Typoperatordefinition $D_{\texttt{tree}}$ aus Abbildung 3.3 definiert.

(1) *Wofür benötigen wir unvollständige Fallunterscheidungen?* Betrachten wir zur Beantwortung dieser Frage den Term

```
if(?node(t),
   0,
   case(t,
        nil : 0,
        leaf : 1,
        node : plus(42, 0))).
```
(3.8)

Die Semantik dieses Terms ist aufgrund der Bedingung `?node(t)` des `if`-Terms vollkommen unabhängig vom `node`-Zweig der inneren Fallunterscheidung. Für eine effiziente symbolische Auswertung ist es daher sinnvoll und korrekt diesen Zweig aus der Fallunterscheidung zu entfernen. Dadurch werden unnötige Vereinfachungen des `node`-Zweigs verhindert. Der Auswertungskalkül vereinfacht daher den Term (3.8) zu der folgenden unvollständigen Fallunterscheidung:

```
if(?node(t),
   0,
   case(t,
        nil : 0,
        leaf : 1)).
```

Unvollständige Fallunterscheidungen entstehen daher während der symbolischen Auswertung eines Terms t, falls der Kontext von Fallunterscheidungen sicherstellt, dass gewisse Zweige der Fallunterscheidungen irrelevant für die Bedeutung des Terms t sind.

(2) *Warum benötigen wir trotz unvollständiger Fallunterscheidungen das Symbol $*$ zur Definition unvollständig-definierter Prozedurdefinitionen?* Unsere intendierte Semantik einer unvollständig-definierten Prozedur f ist, dass jeder Prozeduraufruf $f(t^*)$ für den die Prozedurdefinition von f keine Bedeutung festlegt, durch einen beliebigen Term interpretiert werden darf. Definieren wir

jedoch unvollständig-definierte Prozedurdefinitionen auf Basis unvollständiger Fallunterscheidungen, so ist diese intendierte Semantik nicht mehr gegeben. Betrachten wir hierzu die folgende Prozedurdefinition:

```
function dummy(n, m : tree) : nat ⇐
   case(n,
        leaf : 0,
        node : 1).
```
(3.9)

Die Bedeutung von Prozeduraufrufe der Form

$$\texttt{dummy}(\texttt{nil}, t) \tag{3.10}$$

ist durch die Prozedurdefinition (3.9) nicht festgelegt. Die Bedeutung solcher Prozeduraufrufe ist jedoch auch nicht vollständig beliebig. Vielmehr ist es so, dass alle Prozeduraufrufe der Form (3.10) das gleiche Ergebnis liefern. Das bedeutet, dass für die Prozedurdefinition (3.9) die folgende Gleichung gültig ist:

$$\texttt{dummy}(\texttt{nil}, \texttt{n}) = \texttt{dummy}(\texttt{nil}, \texttt{m}). \tag{3.11}$$

Dies entspricht nicht unserer intendierten Semantik unvollständig-definierter Prozeduren. Wir wollen ja Terme wie $\texttt{dummy}(\texttt{nil}, \texttt{n})$ und $\texttt{dummy}(\texttt{nil}, \texttt{m})$ jeweils durch beliebige aber unbekannte Werte interpretieren. Durch die Gültigkeit der Gleichung sind die Werte der Terme $\texttt{dummy}(\texttt{nil}, \texttt{n})$ und $\texttt{dummy}(\texttt{nil}, \texttt{m})$ nicht mehr vollständig beliebig. Modifizieren wir die Prozedurdefinition von `dummy` zu

```
function dummy(n, m : tree) : nat ⇐
   case(n,
        nil : *,
        leaf : 0,
        node : 1).
```
(3.12)

so ist Gültigkeit der Gleichung (3.11) nicht mehr gegeben. Insgesamt bedeutet das, wir können zwar unvollständig-definierte Prozeduren auf Basis unvollständiger Fallunterscheidungen definieren, jedoch entspricht die Semantik dieser Prozeduren nicht unsere intendierten Semantik. Wir werden daher im nächsten Abschnitt den Rumpf von Prozeduren auf Terme beschränken, die keine unvollständigen Fallunterscheidungen enthalten, um sicherzustellen, dass die Semantik unvollständig-definierter Prozedurdefinitionen unserer intendierten Semantik entspricht. Das bedeutet, dass unvollständige Fallunterscheidungen nur während der symbolischen Auswertung auftreten.

(3) *Ist es notwendig explizite Funktionssymbole* $\texttt{case}^C$ *für unvollständige Fallunterscheidungen einzuführen?* Eine alternative Möglichkeit zur Repräsentation unvollständiger Fallunterscheidungen wäre die irrelevanten Zweige einer Fallunterscheidung mit Hilfe einer Annotation im Term zu markieren:

```
if(?node(t),
   0,
   case^{node}(t,
        nil : 0,
        leaf : 1,
        node : 42)).
```
(3.13)

Dieser Term repräsentiert dann die unvollständige Fallunterscheidung (3.9). Diese unvollständige Fallunterscheidung wird aber beispielsweise auch durch

den folgenden Term repräsentiert:

$$\begin{array}{l}\texttt{if(?node(t),}\\ \quad\texttt{0,}\\ \quad\texttt{case}^{\{\texttt{node}\}}\texttt{(t,}\\ \qquad\texttt{nil : 0,}\\ \qquad\texttt{leaf : 1,}\\ \qquad\texttt{node : 43)).}\end{array} \tag{3.14}$$

Das bedeutet, dass zwei syntaktisch verschiedene Terme die gleiche unvollständige Fallunterscheidung repräsentieren können. Termgleichheit kann dann nicht mehr rein syntaktisch über die Termstruktur definiert werden, sondern muss unter Berücksichtigung der Annotation definiert werden. Dies ist jedoch wenig intuitiv und wir verwenden daher explizite Funktionssymbole zur Repräsentation unvollständiger Fallunterscheidungen.

3.7 Zusätzliche Vereinbarungen

Abschließend wollen wir noch einige Vereinbarungen für Programme und Terme treffen, die die Definitionen der nachfolgenden Kapitel vereinfachen, ohne jedoch die Ausdrucksfähigkeit der Sprache $\mathcal{L}$ zu beschränken.

Wir betrachten nachfolgend ausschließlich Programme, die die Typoperatordefinition $D_{\texttt{nat}}$ (Abbildung 3.1) zur Repräsentation der natürlichen Zahlen und die Prozedurdefinition $D_{>}$ (Abbildung 3.2) zum Vergleich natürlicher Zahlen enthalten. Um durch diese Vereinbarung die Definition von Programmen nicht aufzublähen, verzichten wir jedoch darauf, die Definitionen $D_{\texttt{nat}}$ und $D_{>}$ explizit anzugeben, wenn wir Programme notieren. Weiter notieren wir die Konstruktorgrundterme des Typs `nat` durch die entsprechenden natürlichen Zahlen. So schreiben wir beispielsweise `3` statt `succ(succ(succ(0)))`.

Wir betrachten nachfolgend ausschließlich Terme, die keine Teilterme der Form

$$f(t_1,\ldots,t_n) \quad \text{mit} \quad \begin{array}{l} f \neq \texttt{if} \text{ und } t_i \in \mathcal{T}\!\mathit{erm}_{\texttt{bool}}(P) \text{ für ein} \\ i \in \{1,\ldots,n\}\end{array} \tag{3.15}$$

enthalten. Durch diese Einschränkung gewährleisten wir, dass das Funktionssymbol `if` der einzige Junktor zur Konstruktion boolscher Ausdrücke ist. Insbesondere stellen wir durch die Einschränkung sicher, dass wir ausschließlich Gleichungen von nicht-boolschen Termen betrachten.

Für die Rümpfe von Prozeduren treffen wir eine weitere Vereinbarung. Zur Festlegung dieser Vereinbarung führen wir zunächst den Begriff des $\mathcal{L}$-normalisierten Terms ein.

Definition 3.32 ($\mathcal{L}$-normalisierte Terme)**.** Sei P ein Programm. Ein Term $t \in \mathcal{T}\!\mathit{erm}(P)$ wird als *$\mathcal{L}$-normalisiert* bezeichnet, falls der Term `case`- und `let`-frei ist oder der Term die folgenden Bedingungen erfüllt:

(1) Für alle Teilterme der Form `let` x`:=`s `in` r `end` ist s `case`- und `let`-frei und r $\mathcal{L}$-normalisiert.

(2) Für alle Teilterme der Form $\texttt{case}(b, cons_1 : t_1, \ldots, cons_n : t_n)$ ist b `case`- und `let`-frei und alle Zweige t_i sind $\mathcal{L}$-normalisiert.

□

Wir betrachten dann im weiteren Verlauf dieser Arbeit ausschließlich Programme mit Prozedurdefinitionen $\texttt{function}\ f(x^1_{\tau_1},\ldots,x^n_{\tau_n}) : \tau \Leftarrow R_f$, deren Prozedurrümpfe R_f die folgenden Eigenschaften haben:

- R_f ist ein $\mathcal{L}$-normalisierte Term und enthält Terme der Form $*(\ldots)$ ausschließlich als Zweige von `case`-Termen.
- R_f enthält nur `case`-Terme der Form $\texttt{case}(b, cons_1 : t_1, \ldots, cons_n : t_n)$ wobei $cons_1, \ldots, cons_n$ alle Konstruktoren des entsprechenden Typoperators bezeichnen.

Kapitel 4

Grundlagen des Induktionsbeweisens

Wie wir in Kapitel 2 bereits beschrieben haben, werden Eigenschaften von Programmen im ✓eriFun-System mit Hilfe so genannter Lemmata formuliert. Die Gültigkeit dieser Lemmata wird mit Hilfe von Induktion auf Basis von Sequenzen bewiesen. Die hierbei verwendeten Induktionsaxiome werden durch fundierte Relationenbeschreibungen repräsentiert. In diesem Kapitel definieren wir Syntax und Semantik dieser Lemmata, Sequenzen und Relationenbeschreibungen. Hierzu erweitern wir im wesentlichen die in [46] und [52] definierten Begriffe auf das polymorphe Typsystem des ✓eriFun-Systems.

In Abschnitt 4.1 führen wir zunächst einige grundlegende Begriffe wie Atom, Literal und Quantifikation ein. Aufbauend auf diesen Begriffen definieren wir dann Sequenzen und Lemmata. Anschließend definieren wir in Abschnitt 4.2 unsere polymorphe Erweiterung der Relationenbeschreibungen. Diese Erweiterung ist unkompliziert und erfordert keinen größeren Aufwand.

4.1 Lemmata, Sequenzen und Klauseln

Für ein Programm P bezeichnen wir einen `let`- und `case`-freien Term $a \in \mathcal{T}erm_{\mathtt{bool}}(P) \setminus \{\mathtt{false}\}$ als *Atom* oder *positives Literal*. Die Menge der Atome über P notieren wir mit $\mathcal{A}t(P)$. Ein Atom der Form $l{=}r$ bezeichnen wir als atomare Gleichung und die Menge aller *atomarer Gleichungen* über P bezeichnen wir dann mit $\mathcal{A}t^{=}(P)$. Analog dazu bezeichnen wir ein Atom der Form $?cons(t)$ als *atomaren Strukturtest* und notieren die Menge aller atomaren Strukturtests mit $\mathcal{A}t^{?}(P)$. Einen Term $l \in \mathcal{T}erm_{\mathtt{bool}}(P)$ mit

$$l = \mathtt{false} \quad \text{oder} \quad l = \mathtt{if}(a, \mathtt{false}, \mathtt{true}) \text{ für ein } a \in \mathcal{A}t(P) \setminus \{\mathtt{true}\}$$

nennen wir *negatives Literal* und die Menge aller positiven und negativen Literale über P notieren wir mit $\mathcal{L}it(P)$. Als Junktor verwenden wir das Funktionssymbol $\mathtt{case}^{\{\mathtt{true},\mathtt{false}\}}$ (kurz `if`). Konjunktion, Disjunktion, Negation und Implikation lassen sich dann wie folgt ausdrücken:

$$\begin{aligned}
\mathtt{if}(a, b, \mathtt{false}) &\qquad \text{(Konjunktion)},\\
\mathtt{if}(a, \mathtt{true}, b) &\qquad \text{(Disjunktion)},\\
\mathtt{if}(a, \mathtt{false}, \mathtt{true}) &\qquad \text{(Negation)},\\
\mathtt{if}(a, b, \mathtt{true}) &\qquad \text{(Implikation)}.
\end{aligned}$$

Für Negationen $\texttt{if}(a, \texttt{false}, \texttt{true})$ schreiben wir auch kurz $\neg a$. Das *Komplement* $\bar{l}$ und das Atom $|l|$ eines Literals l definieren wie folgt:

$$\bar{l} \quad := \begin{cases} \texttt{false} & \text{falls } l = \texttt{true}, \\ \texttt{true} & \text{falls } l = \texttt{false}, \\ a & \text{falls } l = \neg a, \\ \neg l & \text{sonst}, \end{cases} \qquad |l| \quad := \begin{cases} \texttt{true} & \text{falls } l = \texttt{true}, \\ \texttt{true} & \text{falls } l = \texttt{false}, \\ a & \text{falls } l = \neg a, \\ l & \text{sonst}. \end{cases}$$

Für endliche Listen boolscher Terme legen wir Konjunktion und Disjunktion wie folgt fest:

$$\begin{aligned} \mathit{AND}(\langle\rangle) &:= \texttt{true}, \\ \mathit{AND}(\langle l \rangle) &:= l, \\ \mathit{AND}(\langle l_1, l_2 \ldots, l_n \rangle) &:= \texttt{if}(l_1, \mathit{AND}(\langle l_2, \ldots, l_n \rangle), \texttt{false}), \end{aligned}$$

$$\begin{aligned} \mathit{OR}(\langle\rangle) &:= \texttt{false}, \\ \mathit{OR}(\langle l \rangle) &:= l, \\ \mathit{OR}(\langle l_1, l_2 \ldots, l_n \rangle) &:= \texttt{if}(l_1, \texttt{true}, \mathit{OR}(\langle l_2, \ldots, l_n \rangle)). \end{aligned}$$

Für $\mathit{AND}(\langle l_1, \ldots, l_n \rangle)$ bzw. $\mathit{OR}(\langle l_1, \ldots, l_n \rangle)$ schreiben wir auch kurz $\mathit{AND}(l_1, \ldots, l_n)$ bzw. $\mathit{OR}(l_1, \ldots, l_n)$. Um Konjunktionen und Disjunktionen für endliche Mengen boolscher Terme zu definieren, setzen wir eine totale Ordnung $>$ auf allen Termen voraus. Für $\{l_1, \ldots, l_n\}$ mit $l_1 > \ldots > l_n$ setzen wir dann

$$\begin{aligned} \mathit{AND}(\{l_1, \ldots, l_n\}) &:= \mathit{AND}(l_1, \ldots, l_n), \\ \mathit{OR}(\{l_1, \ldots, l_n\}) &:= \mathit{OR}(l_1, \ldots, l_n). \end{aligned}$$

Einen Ausdruck der Form

$$\texttt{all}\ \nu_1, \ldots, \nu_n, x^1 : \tau_1, \ldots, x^m : \tau_m\ b \tag{4.1}$$

mit $\nu_1, \ldots, \nu_n \in \mathcal{V}_{\mathcal{T}yp}$, $x^1_{\tau_1}, \ldots, x^m_{\tau_m} \in \mathcal{V}(\Omega(P))$ und $b \in \mathcal{T}erm_{\texttt{bool}}(P)$ nennen wir *Quantifikation über* P. Die Menge aller Quantifikationen über P bezeichnen wir mit $\mathcal{Q}(P)$. Die Typvariablen $\nu_1, \ldots, \nu_n$ bzw. die Termvariablen $x^1_{\tau_1}, \ldots, x^m_{\tau_m}$ der Quantifikation (4.1) nennen wir *gebundene Typ- bzw. Termvariablen* der Quantifikation. Weiter bezeichnen wir eine Typvariable ν als *freie Typvariable* der Quantifikation (4.1), falls $\nu \in \mathit{tv}(b)$ gilt und $\nu_i \neq \nu$ für alle $i \in \{1, \ldots, n\}$. Eine Termvariable x_τ bezeichnen wir als *freie Termvariable* der Quantifikation, falls $x_\tau \in \mathit{fv}(b)$ gilt und $x_\tau \neq x^i_{\tau_i}$ für alle $i \in \{1, \ldots, m\}$. Die Mengen der freien und gebundenen Termvariablen einer Quantifikation q notieren wir dann mit $\mathit{fv}(q)$ und $\mathit{bv}(q)$. Eine Quantifikation q nennen wir *geschlossen*, falls für diese Quantifikation keine freien Typ- und Termvariablen existieren. Für eine Quantifikation der Form $\texttt{all}\ b$, d.h. eine Quantifikation ohne gebundene Typ- und Termvariablen, schreiben wir auch kurz b. Für ein terminierendes Programm P heißt eine geschlossene Quantifikation $\texttt{all}\ \nu^*, x^* : \tau^*\ b$ *gültig* (in Zeichen $P \models \texttt{all}\ \nu^*, x^* : \tau^*\ b$), falls gilt

$$b \equiv_P \texttt{true}$$

Aufbauend auf Quantifikationen definieren wir Lemmata.[1]

[1] Die Syntax von Lemmata im ✓eriFun-System unterscheidet sich von unserer Syntax dadurch, das die Typvariablen nicht quantifiziert werden, d.h. ein Lemma wie

`lemma = _transitive ⇐ all @a, x : @a, y : @a, z : @a if(x=y, if(y=z, x=t, true), true)`

wird in ✓eriFun wie folgt notiert:

`lemma = _transitive ⇐ all x : @a, y : @a, z : @a if(x=y, if(y=z, x=t, true), true).`

Definition 4.1 (Lemma)**.** Sei P ein terminierendes Programm, $l \in \mathcal{S}_{Lem}$ und q eine geschlossene Quantifikation über P. Der Ausdruck

$$\texttt{lemma}\ l \Leftarrow q$$

heißt *Lemma über* P und ist *gültig in* P (in Zeichen $P \models \texttt{lemma}\ l \Leftarrow q$) gdw. $P \models q$ gilt. □

Ähnlich wie für Prozeduren wollen wir uns im Nachfolgenden auf Lemmata beschränken, die nur `case`-Terme der Form $\texttt{case}(b, cons_1 : t_1, \ldots, cons_n : t_n)$ enthalten, wobei $cons_1, \ldots, cons_n$ alle Konstruktoren des entsprechenden Typoperators bezeichnen. Das bedeutet, dass Lemmata immer durch vollständige Fallunterscheidungen definiert sind.

Die Gültigkeit von Lemmata wird in ✓`eriFun` mit Hilfe von Sequenzen nachgewiesen.

Definition 4.2 (Sequenzen)**.** Sei P ein terminierendes Programm, H eine endliche Menge von Literalen, IH eine endliche Menge von Quantifikationen und g ein Term des Typs `bool`. Ein Ausdruck der Form

$$H, IH \vdash g$$

heißt *Sequenz über* P, falls die folgenden Bedingungen erfüllt sind:

(1) Für alle $q \in IH$ gilt, q enthält keine gebundenen Typvariablen.

(2) Für alle $q, q' \in IH$ mit $q \neq q'$ gilt $bv(q') \cap bv(q) = \emptyset$.

H heißt *Hypothesenmenge* der Sequenz, IH *Induktionshypothesenmenge* der Sequenz und g *Goal-Term* der Sequenz. □

Beispiel 4.3. Sei P das Programm aus Beispiel 3.12. Dann sind die folgenden Ausdrücke Sequenzen über P:

$$\{\texttt{l=empty}\}, \emptyset \vdash \texttt{length(add(n,l))>length(l)},$$

$$\{\texttt{l}\neq\texttt{empty}\}, \{\texttt{length(add(n,tl(l)))>length(tl(l))}\} \vdash \texttt{length(add(n,l))>length(l)}.$$

□

Definition 4.4 (Semantik von Sequenzen)**.** Sei P ein terminierendes Programm und $H, IH \vdash g$ eine Sequenz über P. Sei weiter

(1) $\{x^1_{\tau_1}, \ldots, x^n_{\tau_n}\} = fv(H) \cup fv(IH) \cup fv(\mathbf{e}(g))$ und

(2) $\{y^1_{\tau'_1}, \ldots, y^m_{\tau'_m}\} = bv(IH)$.

Die Sequenz $H, IH \vdash g$ heißt gültig in P (in Zeichen $P \models H, IH \vdash g$), falls gilt:

> Für alle P-Interpretationen I, alle I-Typvariablenbelegungen α und alle $d_1 \in [\![\tau_1]\!]^\alpha_{P,I}, \ldots, d_n \in [\![\tau_n]\!]^\alpha_{P,I}$ existieren $e_1 \in [\![\tau'_1]\!]^\alpha_{P,I}, \ldots, e_m \in [\![\tau'_m]\!]^\alpha_{P,I}$, so dass gilt

$$\bigwedge_{h \in H} [\![h]\!]^{\alpha,[x^*/d^*]}_{P,I} = \texttt{true} \wedge \bigwedge_{(\texttt{all}\ z^*:\tau^*\ b) \in IH} [\![b]\!]^{\alpha,[x^*/d^*,y^*/e^*]}_{P,I} = \texttt{true}$$

$$\Longrightarrow [\![g]\!]^{\alpha,[x^*/d^*]}_{P,I} = \texttt{true}$$

□

Es gilt offensichtlich das folgende Lemma:

Lemma 4.5. Sei P ein terminierendes Programm und

$$\texttt{lemma}\, l \;\Leftarrow\; \texttt{all}\; \nu^*, x^* : \tau^*\; b$$

ein Lemma über P. Es gilt dann

$$P \models \texttt{lemma}\, l \;\Leftarrow\; \texttt{all}\; \nu^*, x^* : \tau^*\; b \quad \text{gdw.} \quad P \models \emptyset, \emptyset \vdash b.$$

□

Wie wir in Abschnitt 2.2 beschrieben haben, ist für jedes Lemma

$$\texttt{lemma}\, l \;\Leftarrow\; \texttt{all}\; \nu^*, x^* : \tau^*\; b$$

der Wurzelknoten des durch ✓`eriFun` mit dem Lemma assoziierten Beweisbaums durch eine Sequenz

$$\emptyset, \emptyset \vdash b$$

gegeben. Mit Lemma 4.5 bedeutet das, dass die Gültigkeit des Wurzelknotens des Beweisbaums die Gültigkeit des Lemmas impliziert. Da weiter für jede Beweisregel

$$\frac{seq}{seq_1, \ldots, seq_n} \tag{4.2}$$

des HPL-Kalküls gilt

$$P \models seq_1 \wedge \ldots \wedge P \models seq_n \Longrightarrow P \models seq,$$

folgt somit, dass die Gültigkeit des Lemmas bewiesen ist, falls

(1) die Gültigkeit der während der Konstruktion des Beweisbaums verwendeten Lemmata bewiesen wurde, sowie

(2) die Blätter des Beweisbaums durch Sequenzen mit Goal-Term `true` gegeben sind.

Für die Verwendung von Lemmata und Induktionshypothesen in formalen Beweisen ist es sinnvoll, diese durch Klauselmengen zu repräsentieren. Als *Klausel* bezeichnen wir eine nicht-leere, endliche Menge $\{l_1, \ldots, l_n\}$ von Literalen. Um die freien und gebundenen Typ- und Termvariablen von Lemmata und Induktionshypothesen korrekt in Klauseln zu repräsentieren, führen wir folgende Vereinbarung ein: Lemmata und Induktionshypothesen verwenden ausschließlich *ungestrichene* Typ- und Termvariablen, d.h. keine Variablen wie `@a'` bzw. `x'`. Gebundene Typ- und Termvariablen werden dann in Klauseln immer als *gestrichene* Typ- und Termvariable repräsentiert. Zum Beispiel können wir das Lemma

```
lemma =_transitive ⇐ all @a, x : @a, y : @a, z : @a
  if(x=y, if(y=z, x=t, true), true)
```

durch die folgende Klauselmenge repräsentieren:

$$\{\{\neg(\texttt{x}'_{\texttt{@a}'}=\texttt{y}'_{\texttt{@a}'}), \neg(\texttt{y}'_{\texttt{@a}'}=\texttt{z}'_{\texttt{@a}'}), \texttt{x}'_{\texttt{@a}'}=\texttt{z}'_{\texttt{@a}'}\}\}$$

Die Menge aller gestrichenen Typvariablen eines Terms t bzw. einer Klausel C notieren wir mit $tv'(t)$ bzw. $tv'(C)$ und die Menge aller gestrichenen Termvariablen eines Terms t bzw. einer Klausel C mit $fv'(t)$ bzw. $fv'(C)$. Wir bezeichnen eine Klausel

als *geschlossen*, wenn sie ausschließlich gestrichene Typ- und Termvariablen enthält. Für ein terminierendes Programm P ist eine geschlossene Klausel $\{l_1, \ldots, l_n\}$ gültig in P (in Zeichen $P \models \{l_1, \ldots, l_n\}$), falls gilt

$$OR(l_1, \ldots, l_n) \equiv_P \texttt{true}.$$

Schließlich bezeichnen wir eine Klauselmenge $\{C_1, \ldots, C_n\}$ und eine Quantifikation `all` $\nu^*, x^* : \tau^*\ b$ als äquivalent, falls für alle P-Interpretationen I und alle I-Variablenbelegungen α und a gilt

$$[\![AND(\{OR(C_1), \ldots, OR(C_n)\})]\!]^{\alpha,a}_{P,I} = \texttt{true} \quad \text{gdw.} \quad [\![\sigma_\xi(b)]\!]^{\alpha,a}_{P,I} = \texttt{true}.$$

Hierbei bezeichnen ξ und σ Typ- und Termsubstitutionen, die die Typ- und Termvariable ν^* und x^* in ihre gestrichenen Pendants umbenennen. Wir gehen im Folgenden davon aus, dass mit jeder Quantifikation q eine äquivalente Klauselmenge $\mathcal{C}_q$ assoziiert ist. Diese äquivalente Klauselmenge $\mathcal{C}_q$ kann, wie in [50] skizziert, mit Hilfe der Methoden aus beispielsweise [14], [31] oder [40] berechnet werden. Für ein Lemma

$$lem = (\texttt{lemma}\ l \ \Leftarrow\ q)$$

bezeichnen wir die mit q assoziierte Klauselmenge $\mathcal{C}_q$ auch mit $\mathcal{C}_{lem}$ oder mit $\mathcal{C}_l$.

4.2 Relationenbeschreibungen

Wir verallgemeinern in diesem Abschnitt die in [46] und [50] eingeführten Relationenbeschreibungen auf das polymorphe Typsystem des ✓`eriFun`-Systems. Hierzu definieren wir zunächst polymorphe Relationenbeschreibungen und definieren dann für diese polymorphen Relationenbeschreibungen die Begriffe umbenannte Variante, Typinstantiierung, Domain-Generalisierung, Range-Generalisierung und Separierung.[2] Danach definieren wir die durch Typoperator- und Prozedurdefinitionen definierten polymorphen Relationenbeschreibungen.

Bevor wir polymorphe Relationenbeschreibungen definieren können, benötigen wir den Begriff der *partiellen Termsubstitution*.

Definition 4.6 (Partielle Termsubstitutionen)**.** Sei $\langle \Omega, \Sigma \rangle$ eine Signatur. Eine partielle Abbildung $\delta : \mathcal{V}(\Omega) \mapsto \mathcal{T}erm(\Omega, \Sigma)$ heißt *partielle Termsubstitution*, falls gilt

(1) der Definitionsbereich von δ ist endlich und

(2) $\delta(x_\tau) \in \mathcal{T}erm_\tau(\Omega, \Sigma)$ für alle x_τ des Definitionsbereichs von δ.

Die Menge aller partiellen Termsubstitutionen über $\langle \Omega, \Sigma \rangle$ wird mit $p\mathcal{S}ubst(\Omega, \Sigma)$ bezeichnet. □

Die homomorphe Erweiterung einer partiellen Termsubstitution δ auf `let`-freie Terme definieren wir wie folgt:

$$\delta(f(t_1, \ldots, t_n) := \begin{cases} f(\delta(t_1), \ldots, \delta(t_n)) & \text{falls alle } \delta(t_i) \neq \bot \\ \bot & \text{sonst.} \end{cases}$$

Ähnlich wie für gewöhnliche Termsubstitutionen notieren wir eine partielle Termsubstitution δ mit $dom(\delta) = \{x_1, \ldots, x_n\}$ durch $\{x_1/\delta(x_1), \ldots, x_n/\delta(x_n)\}$. Hierbei ist jeweils aus dem Zusammenhang klar, ob eine partielle oder eine gewöhnliche Termsubstitution gemeint ist.

[2] Für eine ausführliche Beschreibung der Bedeutung von Range-Generalisierung, Domain-Generalisierung und Separierung für Induktionsbeweise verweisen wir auf [46] und [50].

Definition 4.7 (Atomare polymorphe Relationenbeschreibungen)**.** Sei P ein Programm, D eine endliche Liste von Literalen über P und Δ eine endliche Teilmenge von $\mathcal{pSubst}(P)$. Das Paar $A = \langle D, \Delta \rangle$ heißt dann *atomare polymorphe Relationenbeschreibung* über P (oder kurz *atomare Relationenbeschreibung*). Die Literale in D heißen *Domain-Terme* und die partiellen Termsubstitutionen in Δ *Range-Substitutionen*. Die Menge der *relevanten Variablen* von A ist definiert als

$$rv(A) \quad := \quad fv(D) \cup dom(\Delta) \cup fv(rg(\Delta))$$

und die Menge aller *Induktionsvariablen* von A als

$$iv(A) \quad := \quad dom(\Delta).$$

Für eine endliche Liste $\vec{x} = \langle x^1_{\tau_1}, \ldots, x^n_{\tau_n} \rangle$ paarweise verschiedener Termvariablen mit $rv(A) \subseteq \{x^1_{\tau_1}, \ldots, x^n_{\tau_n}\}$, eine P-Interpretation I und eine I-Typvariablenbelegung α ist dann die Relation

$$\rightarrow^{P,I\alpha}_{A,\vec{x}} \subseteq \left([\![\tau_1]\!]^\alpha_{P,I} \times \ldots \times [\![\tau_n]\!]^\alpha_{P,I} \right) \times \left([\![\tau_1]\!]^\alpha_{P,I} \times \ldots \times [\![\tau_n]\!]^\alpha_{P,I} \right)$$

wie folgt definiert:

$$\underbrace{\langle d_1, \ldots, d_n \rangle}_{\vec{d}} \rightarrow^{P,I}_{A,\vec{x}} \langle e_1, \ldots, e_n \rangle$$

gdw.

$$\forall l \in D \,.\, [\![l]\!]^{\alpha,[\vec{x}:\vec{d}]}_{P,I} = \texttt{true} \;\wedge\; \exists \delta \in \Delta \,.\, \forall x^i_{\tau_i} \in dom(\delta) \,.\, e_i = [\![\delta(x^i_{\tau_i})]\!]^{\alpha,[\vec{x}:\vec{d}]}_{P,I}.$$

Die Liste $\vec{x}$ wird als *Dimension* der Relation $\rightarrow^{P,I,\alpha}_{A,\vec{x}}$ bezeichnet. □

Definition 4.8 (Zusammengesetzte polymorphe Relationenbeschreibungen)**.** Sei P ein Programm. Eine endliche Menge $B = \{A_1, \ldots, A_n\}$ atomarer polymorpher Relationenbeschreibungen über P heißt *zusammengesetzte polymorphe Relationenbeschreibung* für P (oder kurz *Relationenbeschreibung*). Die Menge der *relevanten Variablen* von B ist definiert als

$$rv(B) \quad := \quad rv(A_1) \cup \ldots \cup rv(A_n)$$

und die Menge aller *Induktionsvariablen* von B als

$$iv(B) \quad := \quad iv(A_1) \cup \ldots \cup iv(A_n).$$

Für eine endliche Liste $\vec{x} = \langle x^1_{\tau_1}, \ldots, x^n_{\tau_n} \rangle$ paarweise verschiedener Termvariablen mit $rv(B) \subseteq \{x^1_{\tau_1}, \ldots, x^n_{\tau_n}\}$, eine P-Interpretation I und eine I-Typvariablenbelegung α ist die Relation

$$\rightarrow^{P,I,\alpha}_{B,\vec{x}} \subseteq \left([\![\tau_1]\!]^\alpha_{P,I} \times \ldots \times [\![\tau_n]\!]^\alpha_{P,I} \right) \times \left([\![\tau_1]\!]^\alpha_{P,I} \times \ldots \times [\![\tau_n]\!]^\alpha_{P,I} \right)$$

wie folgt definiert:

$$\rightarrow^{P,I,\alpha}_{B,\vec{x}} \quad := \quad \rightarrow^{P,I,\alpha}_{A_1,\vec{x}} \cup \ldots \cup \rightarrow^{P,I,\alpha}_{A_n,\vec{x}} .$$

Die Liste $\vec{x}$ wird als *Dimension* von $\rightarrow^{P,I,\alpha}_{B,\vec{x}}$ bezeichnet. Weiter wird die Menge aller zusammengesetzten polymorphen Relationenbeschreibungen über P mit $\mathcal{Rel}(P)$ notiert. □

Ist im Folgenden die Dimension $\vec{x}$ von $\rightarrow_{B,\vec{x}}^{P,I,\alpha}$ nicht näher beschrieben, so gehen wir davon aus, dass $\vec{x} = \langle x_1, \ldots, x_n \rangle$ mit $\{x_1, \ldots, x_n\} = rv(B)$ gilt. Eine zusammengesetzte polymorphe Relationenbeschreibung B nennen wir *fundiert*, wenn für alle P-Interpretationen und alle Typvariablenbelegungen α die Relation $\rightarrow_{B,\vec{x}}^{P,I,\alpha}$ fundiert ist. Die Menge aller Atome einer Relationenbeschreibung B definieren wir wie folgt:

$$\mathcal{A}t(B) \quad := \quad \bigcup_{\langle D,\Delta\rangle \in B} \big\{ |l| \;\big|\; l \in D \big\} .$$

Definition 4.9 (Umbenannte Variante)**.** Sei $B = \{A_1, \ldots, A_n\}$ eine Relationenbeschreibung mit $A_i = \langle D_i, \Delta_i \rangle$ für alle $i \in \{1, \ldots, n\}$. Seien weiter ξ und σ Typ- und Termsubstitutionen mit

- $|rg(\xi)| = |dom(\xi)|$ und $rg(\xi) \subseteq \mathcal{V}_{\mathcal{T}yp}$ und
- $|rg(\sigma)| = |dom(\sigma)|$ und $rg(\sigma) \subseteq \mathcal{V}(\Sigma(P))$.

Die Relationenbeschreibung $B' = \{A'_1, \ldots, A'_n\}$ mit

- $A'_i = \langle D'_i, \Delta'_i \rangle$,
- $D'_i = \langle \sigma_\xi(l_1), \ldots, \sigma_\xi(l_n) \rangle$ falls $D_i = \langle l_1, \ldots, l_n \rangle$,
- $\Delta'_i = \{\sigma_\xi(\delta) \mid \delta \in \Delta_i\}$[3]

heißt dann *umbenannte Variante* der Relationenbeschreibung B. □

Jede umbenannte Variante einer fundierten Relationenbeschreibung ist trivialerweise ebenfalls fundiert.

Definition 4.10 (Typinstantiierung)**.** Sei $B = \{A_1, \ldots, A_n\}$ eine Relationenbeschreibung mit $A_i = \langle D_i, \Delta_i \rangle$ für alle $i \in \{1, \ldots, n\}$. Sei weiter ξ eine Typsubstitution. Die Relationenbeschreibung $B' = \{A'_1, \ldots, A'_n\}$ mit

- $A'_i = \langle D'_i, \Delta'_i \rangle$, und
- $D'_i = \langle \xi(l_1), \ldots, \xi(l_n) \rangle$ falls $D_i = \langle l_1, \ldots, l_n \rangle$, und
- $\Delta'_i = \{\xi(\delta) \mid \delta \in \Delta_i\}$

heißt dann *Typinstantiierung* der Relationenbeschreibung B. □

Jede Typinstantiierung einer fundierten Relationenbeschreibung ist trivialerweise ebenfalls fundiert.

Definition 4.11 (Range-Generalisierung)**.** Seien $B = \{A_1, \ldots, A_n\}$ und $B' = \{A'_1, \ldots, A'_n\}$ Relationenbeschreibungen mit $A_i = \langle D_i, \Delta_i \rangle$ und $A'_i = \langle D_i, \Delta'_i \rangle$ für alle $i \in \{1, \ldots, n\}$. Die Relationenbeschreibung B' heißt *Range-Generalisierung* von B bzgl. einer Menge $V \subseteq iv(B)$ gdw. für alle $i \in \{1, \ldots, n\}$ gilt

$$\Delta'_i = \{\delta|_{(dom(\delta) \setminus V)} \mid \delta \in \Delta_i\}.$$

Weiter heißt B' *echte* Range-Generalisierung, falls $V \neq \emptyset$ gilt. Ist B' fundiert und existiert keine echte, fundierte Range-Generalisierung von B', so heißt B' *range-maximal*. □

[3] Mit $\sigma_\xi(\delta)$ bezeichnen wir für die partielle Substitution $\delta = \{x_1/t_1, \ldots, x_n/t_n\}$ die zusammengesetzte partielle Substitution $\{\sigma_\xi(x_1)/\sigma_\xi(t_1), \ldots, \sigma_\xi(x_n)/\sigma_\xi(t_n)\}$.

Definition 4.12 (Domain-Generalisierung)**.** Seien $B = \{A_1, \ldots, A_n\}$ und $B' = \{A'_1, \ldots, A'_n\}$ Relationenbeschreibungen mit $A_i = \langle D_i, \Delta_i \rangle$ und $A'_i = \langle D'_i, \Delta_i \rangle$ für alle $i \in \{1, \ldots, n\}$. Die Relationenbeschreibung B' heißt *Domain-Generalisierung* von B bzgl. einer Menge $A \subseteq \mathcal{A}t(B)$ gdw. für alle $i \in \{1, \ldots, n\}$ gilt

$$D'_i = \{d \in D_i \mid |d| \notin A\}.$$

Weiter heißt B' *echte* Domain-Generalisierung falls $A \neq \emptyset$ gilt. Ist B' fundiert und existiert keine echte, fundierte Domain-Generalisierung von B', so heißt B' *domain-maximal.* □

Ist eine Relationenbeschreibung $B = \{\langle D_1, \Delta_1 \rangle, \ldots, \langle D_n, \Delta_n \rangle\}$ range- und domain-maximal, so nennen wir B *maximal.* Wir nennen B *separiert* falls für alle $i, j \in \{1, \ldots, n\}$ mit $i \neq j$ gilt

$$complete_P(D_i) \cap \overline{D_j} \neq \emptyset.$$

Hierbei bezeichnet $\overline{D_j}$ die Menge der negierten Literale von D_j und $complete_P(D_i)$ die kleinste Obermenge von D_i mit

$$\texttt{?}(cons, t) \in complete_P(D_i)$$
$$\Longleftrightarrow$$
$$\left\{\overline{\texttt{?}(cons', t)} \;\middle|\; cons' \neq cons\right\} \subseteq complete_P(D_i).$$

Hierbei bezeichnen *cons* und *cons'* beliebige Konstruktoren eines beliebigen Typoperators des Programms P. Wir nennen eine Relationenbeschreibung *optimal* gdw. sie maximal und separiert ist. Eine Menge M von Relationenbeschreibungen über P nennen wir optimal, wenn alle Relationenbeschreibungen in M optimal sind.

Wir definieren nun die durch Typoperator- und Prozedurdefinitionen definierten Relationenbeschreibungen.

Definition 4.13 (Relationenbeschreibung für Typoperatordefinitionen)**.** Sei P ein Programm und $D \in P$ eine Typoperatordefinition der Form

$$\begin{array}{l}\texttt{structure}\ \zeta[\nu^*] \Leftarrow \\ \quad cons_1(sel_{1,1} : \tau_{1,1}, \ldots, sel_{1,m_1} : \tau_{1,m_1}), \\ \quad \ldots \\ \quad cons_n(sel_{n,1} : \tau_{n,1}, \ldots, sel_{n,m_n} : \tau_{n,m_n}).\end{array}$$

Die mit D assoziierte Relationenbeschreiben B_D ist definiert als

$$\left\{\left\langle\left\{\texttt{?}cons_i(\mathtt{x}_{\zeta[\nu^*]})\right\}, \left\{\left\{\mathtt{x}_{\zeta[\nu^*]}/sel_{i,j}(\mathtt{x}_{\zeta[\nu^*]}) \mid \tau_{i,j} = \zeta[\nu^*]\right\}\right\}\right\rangle \;\middle|\; cons_i \text{ ist reflexiv}\right\}.^4$$

□

Ist für einen Typoperator ζ die Typoperatordefinition D aus dem Zusammenhang klar, so bezeichnen wir die Relationenbeschreibung B_D auch mit B_ζ. Für die Typoperatordefinition von `list` aus Abbildung 3.1 ergibt sich beispielsweise die folgende Relationenbeschreibung:

$$\{\langle\{\texttt{?add(x)}\}, \{\{\texttt{x/tl(x)}\}\}\rangle\}$$

Lemma 4.14. Sei P ein Programm und $D \in P$ eine Typoperatordefinition für ζ entsprechend der Definition 4.13. Die Relationenbeschreibung B_ζ ist dann fundiert. □

[4] Wir bezeichnen ein Funktionssymbol f bezüglich einer Termsignatur Σ als *reflexiv*, falls $\Sigma(f) = \langle \tau_1, \ldots, \tau_n, \tau \rangle$ und ein $i \in \{1, \ldots, n\}$ existiert mit $\tau_i = \tau$.

Da Prozedurrümpfe mit `let`-Termen die Definition der Relationenbeschreibung einer Prozedurdefinition unnötig verkomplizieren, betrachten wir zur Definition der Relationenbeschreibung einer Prozedurdefinition nicht den Prozedurrumpf R_f, sondern den so genannten `let`-*bereinigten Prozedurrumpf*. Unter dem `let`-bereinigten Prozedurrumpf verstehen wir den Term, der durch Ersetzen aller Teilterme der Form $\texttt{let}\ x\texttt{:=}t\ \texttt{in}\ r\ \texttt{end}$ durch $r[x/t]$ aus R_f entsteht. Zur Berechnung des `let`-bereinigten Prozedurrumpfs definieren wir die Funktion $|\cdot|_{\mathcal{P}}$ ($\mathcal{P}$ steht für *purged*):

$$\begin{aligned} |x|_{\mathcal{P}} &:= x \\ |f(t_1,\ldots,t_n)|_{\mathcal{P}} &:= f(|t_1|_{\mathcal{P}},\ldots,|t_n|_{\mathcal{P}}) \\ |\texttt{let}\ x\texttt{:=}t\ \texttt{in}\ r\ \texttt{end}|_{\mathcal{P}} &:= |r\,[x/|t|_{\mathcal{P}}]\,|_{\mathcal{P}} \end{aligned}$$

Es gilt das folgende Lemma:

Lemma 4.15. Sei P ein terminierendes Programm und $t \in \mathcal{Term}(P)$. Für alle P-Interpretationen I und alle I-Variablenbelegungen α und a gilt

$$[\![t]\!]^{\alpha,a}_{P,I} = [\![|t|_{\mathcal{P}}]\!]^{\alpha,a}_{P,I}.$$

□

Weiter benötigen wir zur Definition der Relationenbeschreibung einer Prozedurdefinition die Bedingungen, unter denen die rekursiven Aufrufe in R_f stattfinden. Wir definieren hierzu für einen `let`-freien Term t und eine Position $\pi \in \mathcal{Pos}(t)$ die endliche Liste $cond(\pi, t)$:

$$\begin{aligned} cond(\epsilon, t) &:= \langle\rangle, \\ cond(i.\pi, f(t_1,\ldots,t_n)) &:= cond(\pi, t_i), \quad \text{falls } f \notin \Sigma^{\text{case}}(P), \\ cond(i.\pi, \texttt{case}(b, cons_1 : t_1, \ldots, cons_n : t_n)) &:= \\ &\begin{cases} cond(\pi, b) & \text{falls } i = 1, \\ \texttt{?}(cons_{i-1}, b).cond(\pi, t_{i-1}) & \text{falls } i > 1. \end{cases} \end{aligned}$$

Wir definieren dann die Relationenbeschreibung einer Prozedurdefinition wie folgt:

Definition 4.16 (Relationenbeschreibung für Prozedurdefinitionen)**.** Sei P ein Programm und $D \in P$ eine Prozedurdefinition der Form

$$\texttt{function}\ f(x^1 : \tau_1, \ldots, x^n : \tau_n) : \tau \ \Leftarrow\ R_f.$$

Sei weiter $\Pi = \{\pi \in \mathcal{Pos}(|R_f|_{\mathcal{P}}) \mid |R_f|_{\mathcal{P}}|_\pi = f(t^\pi_1,\ldots,t^\pi_n)\}$ und für alle $\pi \in \Pi$ sei die partielle Termsubstitution δ_π gegeben als $\{x^1_{\tau_1}/t^\pi_1, \ldots, x^n_{\tau_n}/t^\pi_n\}$. Die mit D assoziierte Relationenbeschreibung B_D ist dann gegeben als

$$\{\langle cond(\pi, |R_f|_{\mathcal{P}}), \{\delta_{\pi'} \mid cond(\pi', |R_f|_{\mathcal{P}}) = cond(\pi, |R_f|_{\mathcal{P}}) \text{ für } \pi' \in \Pi\}\rangle \mid \pi \in \Pi\}\,.$$

□

Ist für eine Prozedur f die Prozedurdefinition D aus dem Zusammenhang klar, so bezeichnen wir die Relationenbeschreibung B_D auch mit B_f. Für die Prozedurdefinition `>` aus Abbildung 3.2 ergibt sich beispielsweise die folgende Relationenbeschreibung:

$$\{\langle\{\texttt{?succ(x)}, \texttt{?succ(y)}\}, \{\{\texttt{x/pred(x)}, \texttt{y/pred(y)}\}\}\rangle\}$$

Lemma 4.17. Sei P ein terminierendes Programm und $D \in P$ eine Prozedurdefinition für f entsprechend der Definition 4.16. Die Relationenbeschreibung B_f ist dann fundiert. □

Relationenbeschreibung werden in ✓eriFun zur Berechnung der Induktionsformeln von Induktionsbeweisen verwendet. Hierbei nutzt das System den im zweiten Teil dieser Arbeit definierten Auswertungskalkül, um möglichst einfache Hypothesen und Induktionshypothesen zu erzeugen. Das im ✓eriFun-System implementierte Verfahren zur Berechnung der Induktionsformeln ist in Anhang B ausführlich beschrieben. Für die folgenden Kapitel ist es jedoch ausreichend anzunehmen, dass die Induktionsformeln mit Hilfe der nachfolgend definierten Menge $\mathit{Ind}(R, g)$ erzeugt werden.

Definition 4.18 (Induktionsformeln)**.** Sei P ein terminierendes Programm, R eine Relationenbeschreibung über P und g ein Term des Typs `bool`. Die Relationenbeschreibung R heißt *zulässige Relationenbeschreibung* für g gdw. $rv(R) \subseteq fv(g)$ gilt. Ist R eine zulässige Relationenbeschreibung für g, so ist die Menge $\mathit{Ind}(R, g)$ der *Induktionsformeln* für g bzgl. R definiert als die kleinste Menge von Sequenzen über P, die die folgenden Bedingungen erfüllt:

(1) $H, \emptyset \vdash g \in \mathit{Ind}(R, g)$ falls $H \subseteq \mathcal{L}it(P)$ und

  (a) $\forall\, h \in H$ gilt $|h| \in \mathcal{A}t(R)$,

  (b) $h \in H \iff \overline{h} \notin H$,

  (c) $\forall\, \langle H', \Delta' \rangle \in R$ gilt $H' \not\subseteq H$.

(2) $H, IH \vdash g \in \mathit{Ind}(R, g)$ falls $IH = \bigcup_{i=1}^{n} \{\texttt{all}\ z_{m_i+1}, \ldots, z_{m_{i+1}}\ \delta_i'(g)\}$ und

  (a) $\langle H, \{\delta_1, \ldots, \delta_n\} \rangle \in R$,

  (b) $\{y_{m_i+1}, \ldots, y_{m_{i+1}}\} = fv(g) \setminus dom(\delta_i)$ mit $m_0 = 0$,

  (c) $z_j = Var(\{fv(g) \cup \{z_0, \ldots, z_{j-1}\}\})$,

  (d) $\delta_i' \in \mathcal{S}ubst(P)$ mit

$$\delta_i'(x) = \begin{cases} z_j & \text{falls } x = y_j \text{ und } j \in \{m_i + 1, \ldots, m_{i+1}\}, \\ \delta_i(x) & \text{sonst.} \end{cases}$$

□

Es gilt das folgende Lemma (siehe hierzu auch [50]):

Lemma 4.19. Sei P ein terminierendes Programm, g ein Term des Typs `bool` und R eine zulässige, fundierte Relationenbeschreibung über P. Die Sequenzen der Menge $\mathit{Ind}(R, g)$ sind genau dann in P gültig, wenn für alle P-Interpretationen I und alle I-Variablenbelegungen α und a gilt

$$[\![\mathsf{e}(g)]\!]_{P,I}^{\alpha,a} = \texttt{true}.$$

□

Für Relationenbeschreibungen B und B' ist im Allgemeinen nicht entscheidbar, ob für alle P-Interpretationen I und alle I-Typvariablenbelegungen α die Relationen $\rightarrow_{B,\vec{x}}^{P,I,\alpha}$ Teilrelationen der entsprechenden Relationen $\rightarrow_{B',\vec{x}}^{P,I,\alpha}$ sind, d.h. ob gilt

$$\rightarrow_{B,\vec{x}}^{P,I,\alpha} \quad \subseteq \quad \rightarrow_{B',\vec{x}}^{P,I,\alpha} .$$

Wir definieren daher eine polymorphe Erweiterung der Subsumptionsrelation aus [50], um Relationenbeschreibungen auf Basis einer entscheidbaren Relation vergleichen zu können.

Definition 4.20 (Subsumption von Relationenbeschreibungen)**.** Sei P ein terminierendes Programm und seien $B, B' \in \mathcal{Rel}(P)$. B' *subsumiert* B (in Zeichen $B \sqsubseteq B'$) gdw. für alle $\langle D, \Delta\rangle \in B$ ein $\langle D', \Delta'\rangle \in B'$ und für alle $\delta \in \Delta$ ein $\delta' \in \Delta'$ existiert, so dass $D' \subseteq D$ und $\delta' \subseteq \delta$ gilt. □

Die Subsumptionsrelation $\sqsubseteq$ ist entscheidbar und es gilt folgendes Lemma:

Lemma 4.21. Sei P ein Programm. Für alle Relationenbeschreibungen B und B' aus $\mathcal{Rel}(P)$ mit $B \sqsubseteq B'$ sowie für alle P-Interpretationen I und alle I-Typvariablenbelegungen α ist $\rightarrow^{P,I,\alpha}_{B,\vec{x}} \subseteq \rightarrow^{P,I,\alpha}_{B',\vec{x}}$. □

Im ✓eriFun-System ist mit jeder Prozedur f eines terminierenden Programms eine nicht-leeren, endlichen Menge $\{B^1_f, \ldots, B^n_f\}$ von Relationenbeschreibungen assoziiert, die die folgenden Bedingungen erfüllt:

(1) $B_f \sqsubseteq B^i_f$.

(2) B^i_f ist fundiert.

(3) $B^i_f \not\sqsubseteq B^j_f$ für alle $j \in \{1, \ldots, n\}$ mit $i \neq j$.

(4) B^i_f ist separiert.

Wir nennen die Relationenbeschreibungen dieser Menge die *generalisierten Relationenbeschreibungen* der Prozedur f und bezeichnen die Menge $\{B^1_f, \ldots, B^n_f\}$ mit $grds_P(f)$ (*grds* steht für *generalized relation description set*). Falls P aus dem Zusammenhang klar ist, schreiben wir auch kurz $grds(f)$. Die Existenz einer solchen Menge für jede Prozedur f in P ist dadurch sichergestellt, dass wir immer $grds_P(f) = \{B_f\}$ setzen können.

Teil II

Auswertungskalkül

Kapitel 5

Anforderungen

Wir kommen nun zum Hauptteil der vorliegenden Arbeit: Die Definition des Auswertungskalküls. Beim Auswertungskalkül handelt es sich, wie bereits erwähnt, um einen Beweiskalkül für Gleichheitsbeweise. Er ist für Beweisprobleme optimiert, die typischerweise bei der Verifikation funktionaler Programme auftreten. Die Implementierung dieses Kalküls bildet die voll-automatische Beweiskomponente und damit den Kern des ✓eriFun-Systems. Die Definition des Auswertungskalküls erstreckt sich über mehrere Kapitel. Ziel dieses Kapitels ist es die Anforderungen zusammenzufassen, die an den Auswertungskalkül gestellt werden (Abschnitt 5.1) und die Umsetzung dieser Anforderungen zu skizzieren (Abschnitt 5.2).

5.1 Beschreibung der Anforderungen

Der Auswertungskalkül muss die folgenden – zum Teil gegensätzlichen – Anforderungen erfüllen:

- **Korrektheit:** Die Korrektheit ist die Mindestanforderung, die an jeden Logikkalkül zu stellen ist. Bei den nachfolgenden Definitionen der einzelnen Auswertungsregeln in den Kapiteln 8 - 13 werden wir die Korrektheit der jeweiligen Regel, sofern die Korrektheit nicht offensichtlich ist, informell begründen.
- **Terminierung:** Da es sich beim ✓eriFun-System um ein interaktives Beweissystem handelt, muss die Berechnung von Herleitungen im Auswertungskalkül terminieren. Die Terminierung versuchen wir durch verschiedene Heuristiken zu gewährleisten. Hierbei steht insbesondere die Nützlichkeit der Ergebnisse der symbolischen Auswertung für nachfolgende Beweisschritte im Vordergrund. Obwohl wir nur terminierende Programme betrachten, können unsere Heuristiken die Terminierung nicht sicherstellen. Es ist daher erforderlich, die maximale Länge einer Auswertung zusätzlich zu begrenzen. Wir kommen hierauf im dritten Teil der Arbeit zu sprechen.
- **Determiniertheit:** Dies wird gefordert, um Implementierungsunabhängigkeit zu garantieren. Folglich darf zu jedem Zeitpunkt höchstens eine Regel des Auswertungskalküls anwendbar sein. Dies wird dadurch realisiert, dass die Anwendbarkeit der Regeln in einer festen Reihenfolge überprüft werden und die erste anwendbare Regel angewendet wird. Weiter muss jede Regel selbst deterministisch sein. Einige der Regeln des Auswertungskalkül sind auf Basis von Mengen definiert. So wählt beispielsweise die in Kapitel 11 definierte Regel „*Affirmative assumption*“ einen beliebigen Matcher aus der Menge aller Matcher aus. Diese Regeln sind ohne weitere Vereinbarungen nicht deterministisch. Sofern bei der Definition der Regeln nichts weiter gesagt wird, gehen

wir daher davon aus, dass auf der entsprechenden Menge eine totale Ordnung definiert ist und die Elemente in der durch diese totale Ordnung definierte Reihenfolge überprüft bzw. angewendet werden. Wir gewährleisten dadurch die Determiniertheit auch dieser Regeln.

- **Effizienz:** Der Auswertungskalkül muss effizient sein. Effizienz ist eine Anforderung, die jedes Software-System – insbesondere wenn es sich um ein interaktives System handelt – erfüllen muss. Für Beweissysteme bedeutet Effizienz, dass der Suchraum entsprechend eingeschränkt werden muss. Hierzu definieren wir für die einzelnen Auswertungsregeln verschiedene Heuristiken, die die Anwendung der Auswertungsregeln entsprechend einschränken.

- **Mächtigkeit:** Je mehr Sequenzen existieren, die mit dem Auswertungskalkül vollautomatisch bewiesen werden können, desto weniger Interaktionen von Seiten des Benutzers sind nötig, um eine Sequenz zu beweisen. Dementsprechend weniger Expertenwissen muss ein Benutzer in einen Beweis investieren und entsprechend effektiv und komfortabel ist die Arbeit des Benutzers mit dem ✓eriFun-System. Wir sind also bemüht, einen möglichst mächtigen Auswertungskalkül zu definieren. Da die Berechnung der Herleitungen im Auswertungskalkül terminieren soll, kann man jedoch aufgrund der Unentscheidbarkeit der Prädikatenlogik keinen vollständigen Kalkül definieren [17, 41].

- **Nützlichkeit:** Der Auswertungskalkül soll insgesamt als Ergebnis seiner Berechnungen einen gegenüber dem ursprünglichen Term vereinfachten Term zurückliefern. Hierbei verstehen wir unter einem vereinfachten Term nicht notwendigerweise einen kleineren oder kürzeren Term, sondern vielmehr einen Term, der aus beweistechnischer Sicht ein einfacheres Problem darstellt.

5.2 Umsetzung der Anforderungen

Insbesondere Effizienz und Mächtigkeit sind konkurrierende Anforderungen. Eine Vergrößerung des Suchraums erhöht zwar die Mächtigkeit, da mehr Sequenzen bewiesen werden können, die Effizienz reduziert sich aber entsprechend. Es ist ein zentraler Beitrag dieser Arbeit, einen guten Kompromiss zwischen Effizienz und Mächtigkeit zu definieren. Die folgenden Aspekte des Auswertungskalküls sind hierbei besonders zu beachten:

- **Reihenfolge:** Die Reihenfolge der Regeln des Auswertungskalküls spielt eine entscheidende Rolle für die Effizienz. Unnötige Regelanwendungen und damit unnötige Auswertungen können dadurch verhindert werden, dass die Regeln sinnvoll angeordnet sind.

- **Anwendungsbedingungen:** Die Anwendungsbedingungen der Regeln können zusätzliche Suchräume einführen. So existieren z.B. Regeln, die den Auswertungskalkül rekursiv verwenden, um zu überprüfen, ob gewisse Anwendungsbedingungen erfüllt sind. Die Überprüfung solcher Anwendungsbedingungen ist teuer und muss durch entsprechende Kontrollmechanismen eingeschränkt werden.

- **Auswertungsumgebung:** Der Auswertungskalkül wertet einen Term t bzgl. einer bestimmten *Auswertungsumgebung* (kurz *A-Umgebung* genannt) aus. Diese A-Umgebung enthält alle Informationen, die wir zur Auswertung des Terms benötigen. Unter anderem enthält die A-Umgebung die Hypothesen- und die Induktionshypothesenmenge der Sequenz des zu vereinfachenden

Goal-Terms. Weiter legt die A-Umgebung fest, welche vom ✓eriFun-System bereits bewiesenen Lemmata zur symbolischen Auswertung verwendet werden dürfen. Je mehr Lemmata verwendet werden dürfen, desto größer wird der Suchraum und desto ineffizienter wird die Auswertung eines Terms.

- **Auswertungsannotationen:** Die Informationen einer A-Umgebung reichen alleine nicht aus, um Terme effizient und zielgerichtet auszuwerten. Es ist vielmehr notwendig, die Terme mit Zusatzinformationen – so genannten *Annotationen* – an zu reichern und so die Auswertung der Terme zu steuern. Die Annotationen, die wir zur symbolischen Auswertung verwenden, heißen *Auswertungsannotationen* (oder kurz *A-Annotationen*). Sie legen unter anderem fest, ob Induktionshypothesen und Lemmata auf einen Term angewendet werden dürfen und ob Prozeduraufrufe in einem Term geöffnet werden dürfen. Verbieten die A-Annotationen eines Terms die Anwendung von Induktionshypothesen und Lemmata bzw. das Öffnen von Prozeduren, so kann man sich die Überprüfung der entsprechenden Regeln des Auswertungskalküls sparen und so die Überprüfung teurer Anwendungsbedingungen vermeiden. Für die Effizienz des Auswertungskalküls ist es daher von großer Bedeutung, auf welche Werte die Annotationen der Terme gesetzt werden.

Kapitel 6

Die Steuerinformationen

Wir führen in diesem Kapitel die bereits in Abschnitt 5.1 erwähnten A-Umgebungen und A-Annotationen ein. Eine formale Definition dieser Begriffe erfolgt in Abschnitt 6.1. Die Verwendung von A-Umgebungen und A-Annotationen erläutern wir in Abschnitt 6.2.

6.1 A-Umgebungen, A-Annotationen und A-Terme

Wir bezeichnen das Tupel aller Informationen, die wir zur symbolischen Auswertung eines Terms benötigen, als A-Umgebung. Bevor wir A-Umgebungen formal definieren, wollen wir die benötigten Informationen benennen und kurz erläutern. Wir gehen hierbei davon aus, dass wir den Term t der Sequenz

$$H, IH \vdash t \tag{6.1}$$

symbolisch auswerten möchten. Eine A-Umgebung setzt sich dann aus folgenden Komponenten zusammen:

- **Das aktuelle Programm:** Um die Funktionssymbole, die in t vorkommen, korrekt interpretieren zu können, muss das Programm P bekannt sein bzgl. dessen t definiert ist.
- **Die auszuwertenden Prozeduren:** Um die Auswertung von Prozeduren präzise steuern zu können, ist es notwendig, die Prozeduren zu kennen, deren Prozeduraufrufe ausgewertet werden dürfen. Diese Prozeduren werden in der A-Umgebung durch eine Termsignatur $\Sigma \subset \Sigma^{\text{proc}}(P)$ repräsentiert. Ist eine Prozedur nicht in Σ enthalten, so wird kein Prozeduraufruf dieser Prozedur ausgewertet. Für Beispiel 2.1 bedeutet dies, dass Σ mindestens die Prozeduren `mult`, `dbl` und `plus` enthalten muss, da für diese Prozeduren Funktionaufrufe ausgewertet werden.
- **Die als gültig vorauszusetzenden Quantifikationen:** Zur symbolischen Auswertung des Goal-Terms t der Sequenz (6.1) werden die Induktionshypothesen und die vom ✓`eriFun`-System bereits verifizierten Lemmata verwenden. Während der symbolischen Auswertung werden die Induktionshypothesen und die Lemmata in einer A-Umgebung durch eine entsprechende Multimenge $\mathcal{Q}$ von Quantifikationen repräsentiert. Die Notwendigkeit einer Multimenge erläutern wir in Kapitel 11.
- **Die aktuellen lokalen Hypothesen:** Zur symbolischen Auswertung eines Zweigs t_i eines `case`-Terms $\texttt{case}(b, \ldots, cons : t, \ldots)$ dürfen wir den boolschen

Term ?(*cons*, *b*) als gültig voraussetzen. Wir bezeichnen ?(*cons*, *b*) dann auch als lokale Hypothese zur Auswertung von t. Das bedeutet, dass wir zur Auswertung der einzelnen Zweige eines `case`-Terms unterschiedliche lokale Hypothesen verwenden. Eine A-Umgebung muss daher die Menge H_{L} der aktuellen lokalen Hypothesen enthalten.

- **Die aktuellen globalen Hypothesen:** Zur symbolischen Auswertung des Goal-Terms der Sequenz (6.1) dürfen die Hypothesen der Sequenz verwendet werden. Da die Hypothesen der Sequenz für alle Teilterme des Goal-Terms der Sequenz gültig sind, bezeichnen wir diese als *globale* Hypothesen.
- **Die aktuellen Instanzhypothesen:** Zur symbolischen Auswertung des Goal-Terms t der Sequenz (6.1) ist eine dritte Hypothesenmenge H_{I} notwendig. Hierbei handelt es sich um eine Teilmenge der Menge der lokalen Hypothesen H_{L}. Die Hypothesen der Menge H_{I} werden verwendet, um während der symbolischen Auswertung Instanzen der Quantifikationen aus $\mathcal{Q}$ zu bilden. Welche der lokalen Hypothesen in H_{I} enthalten sind, wird durch das nachfolgend eingeführte Instanz-Flag bestimmt. Die genaue Bedeutung, Notwendigkeit und Verwendung der Instanzhypothesenmenge und des Instanz-Flags wird in Kapitel 11 beschrieben.
- **Die aktuellen Variablenbindungen:** Zur symbolischen Auswertung des Rumpfes r eines Terms `let` x`:=`t `in` r `end` wird die Variablenbindung x=t benötigt. Die aktuellen Variablenbindungen werden in einer A-Umgebung durch eine Termsubstitution δ repräsentiert. Hierbei enthält die Termsubstitution für jede Variablenbindung x=t ein Paar der Form x/t. Beispielsweise enthält die Termsubstitution δ zur Auswertung des Rumpfs des `let`-Terms `let x:=0 in x` = `succ(y) end` das Paar `x/0`.
- **Die aktuelle Kontextmenge:** Mit Hilfe der oben beschriebenen Termsignatur Σ der auszuwertenden Prozeduren können wir lediglich die Auswertung aller Prozeduraufrufe einer Prozedur f sperren. Es ist nicht möglich lediglich bestimmte Prozeduraufrufe zu sperren. A-Umgebungen verwalten daher eine so genannte Kontextmenge C, mit deren Hilfe bestimmte Prozeduraufrufe einer Prozedur gesperrt werden können. Die Kontextmenge steht in engem Zusammenhang mit den nachfolgend definierten Labels der A-Annotationen. Eine genaue Erklärung der Funktionsweise der Kontextmenge verschieben wir daher auf die nachfolgende Besprechung der A-Annotationen.
- **Die kommutativen und assoziativen Prozeduren:** Prozeduraufrufe von kommutativen oder assoziativen Prozeduren werden während der symbolischen Auswertung speziell behandelt. So kann der Auswertungskalkül beispielsweise die Gleichung `plus(x,y)=plus(y,x)` mittels der Reflexivität von = zu `true` vereinfachen, sofern bekannt ist, dass die Prozedur `plus` kommutativ ist. Ein anderes Beispiel ist die Gleichung `xor(x,xor(y,z))=xor(y,xor(z,x))`. Diese Gleichung kann vom Auswertungskalkül ebenfalls zu `true` ausgewertet werden, wobei jedoch die Kommutativität und die Assoziativität der Prozedur `xor` bekannt sein muss. Kommutativität und Assoziativität von Prozeduren wird in ✓`eriFun` durch den Nachweis der Gültigkeit der entsprechenden Lemmata (siehe Definition 3.31) bewiesen. Die kommutativen bzw. die assoziativen Prozeduren werden dann während der symbolischen Auswertung durch die Termsignatur Σ_{C} und Σ_{A} repräsentiert.
- **Die anzuwendenden Regeln:** Es ist sinnvoll, in gewissen Situationen nur einen Teil der Auswertungsregeln zuzulassen. So wollen wir beispielsweise gelegentlich die Anwendung bestimmter Regeln zur Auswertung von Prozeduraufrufen verhindern. Um die Reihenfolge der Regeln während der symbolischen

Auswertung festzulegen, ist jede Regel mit einer eindeutigen Nummer assoziiert. Diese Nummer können wir auch dazu verwenden, die Anwendung der Regeln zu kontrollieren. Hierzu verwaltet eine A-Umgebung die Menge $\mathcal{R}$ der Regelnummern, deren Regeln verwendet werden dürfen.

Damit haben wir alle Informationen benannt und erläutert, die wir zur symbolischen Auswertung eines Terms benötigen. Diese Informationen fassen wir nun formal zu einer *A-Umgebung* zusammen.

Definition 6.1 (A-Umgebung)**.** Sei P ein terminierendes Programm. Seien weiter

- Σ, Σ_{C} und Σ_{A} Teilsignaturen von $\Sigma^{\mathrm{proc}}(P)$,
- $H_{\mathrm{L}}, H_{\mathrm{G}}$ und H_{I} endliche Teilmengen von $\mathcal{L}\mathit{it}(P)$ mit $H_{\mathrm{I}} \subseteq H_{\mathrm{L}}$,
- $\mathcal{Q}$ eine endliche Multimenge von Quantifikationen von P,
- C eine Teilmenge von $dom(\Sigma^{\mathrm{proc}}(P)) \cup \{\top\}$ mit $\top \in C$,
- $\mathcal{R}$ eine Teilmenge von $\{1, \ldots, R_{\max}\}$[1] und
- δ eine Substitution aus $\mathcal{S}\mathit{ubst}(P)$.

Eine *Auswertungsumgebung* (oder kurz *A-Umgebung*) von P ist dann ein Tupel der Form

$$\mathcal{U} = \langle P, \Sigma, \Sigma_{\mathrm{C}}, \Sigma_{\mathrm{A}}, \mathcal{Q}, H_{\mathrm{L}}, H_{\mathrm{G}}, H_{\mathrm{I}}, C, \mathcal{R}, \delta \rangle.$$

Die Komponenten von $\mathcal{U}$ werden wie folgt bezeichnet:

- Σ = *Termsignatur der auszuwertenden Prozeduren* der A-Umgebung.
- Σ_{C} = *Termsignatur der kommutativen Prozeduren* der A-Umgebung.
- Σ_{A} = *Termsignatur der assoziativen Prozeduren* der A-Umgebung.
- H_{L} = *lokale Hypothesenmenge* der A-Umgebung.
- H_{G} = *globale Hypothesenmenge* der A-Umgebung.
- H_{I} = *Instanzhypothesenmenge* der A-Umgebung.
- δ = *Variablenbindung* der A-Umgebung..
- C = *Kontextmenge* der A-Umgebung.
- $\mathcal{R}$ = *Menge der zulässigen Regeln* der A-Umgebung.

□

Da die Informationen der A-Umgebung nicht ausreichen, um einen Term effizient auszuwerten, müssen wir Terme mit Annotationen markieren, um so die Auswertung von Termen zusätzlich kontrollieren zu können. Hierzu führen wir zunächst allgemein den Begriffe der annotierten Terme ein. Unsere Definition annotierter Terme orientiert sich dabei an den Begriffsbildungen für annotierte Terme aus [26].

Definition 6.2 (Annotierte Terme)**.** Sei P ein terminierendes Programm und Φ eine beliebige nicht-leere Menge. Für einen Typ $\tau \in \mathcal{T}\mathit{yp}(\Omega(P))$ ist die Menge $\Phi\text{-}\mathcal{T}\mathit{erm}_{\tau}(P)$ der mit Φ *annotierten Terme von Typ* τ *über* P definiert als die kleinste Menge mit

(1) $x_{\tau} \in \Phi\text{-}\mathcal{T}\mathit{erm}_{\tau}(P)$, falls $x_{\tau} \in \mathcal{V}(\Omega(P))$,

[1] $R_{\max}$ bezeichnet die Regelnummer der letzten Regel des Auswertungskalküls.

(2) $f^A_\tau(t_1, \ldots, t_n) \in \Phi\text{-}\mathcal{Term}_\tau(P)$, falls $A \in \Phi$, $t_1 \in \Phi\text{-}\mathcal{Term}_{\tau_1}(P), \ldots, t_n \in \Phi\text{-}\mathcal{Term}_{\tau_n}(P)$, $\Sigma(P)(f) = \langle \tau'_1, \ldots, \tau'_n, \tau'_{n+1} \rangle$ und eine Typsubstitution $\xi \in \mathcal{Subst}(\Omega)$ existiert mit $\tau = \xi(\tau'_{n+1})$ und $\tau_1 = \xi(\tau'_1), \ldots, \tau_n = \xi(\tau'_n)$,

(3) $\texttt{let}^A\ x_{\tau'}\texttt{:=}t\ \texttt{in}\ r\ \texttt{end} \in \Phi\text{-}\mathcal{Term}_\tau(P)$, falls $A \in \Phi$, $t \in \Phi\text{-}\mathcal{Term}_{\tau'}(P)$, $r \in \Phi\text{-}\mathcal{Term}_\tau(P)$ und $x_{\tau'} \in \mathcal{V}(\Omega(P))$.

Mit $\Phi\text{-}\mathcal{Term}(P)$ wird die Menge aller mit Φ annotierten Terme bezeichnet. Für einen annotierten Term t notieren wir mit $\mathsf{e}(t)$ den Term, der aus t entsteht, wenn alle Annotationen in t gestrichen werden:

$$\begin{aligned} \mathsf{e}(x_\tau) &:= x_\tau, \\ \mathsf{e}(f^A(t_1, \ldots, t_n)) &:= f(\mathsf{e}(t_1), \ldots, \mathsf{e}(t_n))), \\ \mathsf{e}(\texttt{let}^A\ x_\tau\texttt{:=}t\ \texttt{in}\ r) &:= \texttt{let}\ x_\tau\texttt{:=}\mathsf{e}(t)\ \texttt{in}\ \mathsf{e}(r). \end{aligned}$$

Eine Abbildung $\sigma : \mathcal{V}(\Omega(P)) \to \Phi\text{-}\mathcal{Term}(P)$ mit $\sigma(x_\tau) \neq x_\tau$ für endlich viele $x_\tau \in \mathcal{V}(\Omega(P))$ und $\sigma(x_\tau) \in \Phi\text{-}\mathcal{Term}_\tau(P)$ für alle $x_\tau \in \mathcal{V}(\Omega(P))$ nennen wir *annotierte Termsubstitution*. Die Menge aller mit Φ annotierten Termsubstitutionen über P bezeichnen wir mit $\Phi\text{-}\mathcal{Subst}(P)$. □

Es lassen sich alle Begriffe und Schreibweisen für Terme und Termsubstitutionen wie beispielsweise Position, Teilterm, Teiltermersetzung, freie und gebundene Termvariablen, etc. aus Kapitel 3 leicht auf annotierte Terme und annotierte Termsubstitutionen übertragen. Es gilt da folgende Lemma

Lemma 6.3. Sei P ein Programm und Φ eine beliebige nicht-leere Menge. Es gilt dann:

$$\mathsf{e}(\Phi\text{-}\mathcal{Term}_\tau(P)) = \mathcal{Term}_\tau(P).$$

□

Nachdem wir nun annotierte Terme definiert haben, wollen wir die Annotationen definieren, die für eine effiziente symbolische Auswertung benötigt werden. Diese Annotationen fassen wir zu einem Tupel zusammen. Dieses Tupel bezeichnen wir als Auswertungsannotation (oder kurz A-Annotation). Bevor wir A-Annotationen formal definieren, wollen wir – ähnlich wie bei A-Umgebungen – die Komponenten einer A-Annotation benennen und kurz erläutern:

- **Die Polarität:** Um boolsche Terme effizient auszuwerten, enthält eine A-Annotation eine *Polarität* p. Es existieren drei mögliche Polaritäten: positiv ($\oplus$), negativ ($\ominus$) und neutral ($\odot$). Eine positive Polarität bedeutet, dass der Term zu `true` ausgewertet werden soll und daher die Überprüfung von Auswertungsregeln, die den Term zu `false` vereinfachen würden, nicht sinnvoll ist. Entsprechendes gilt für negative Polarität. Eine neutrale Polarität bedeutet, dass der Term beliebig auszuwerten ist und keinerlei Einschränkungen bezüglich des gewünschten Wahrheitswerts existieren. Hierbei ist insgesamt folgendes zu beachten: Die Polarität ist lediglich eine heuristische Information, um die Auswertung von boolschen Termen in bestimmten Situationen zu kontrollieren. Aufgrund der Polarität wird in keinem Fall die Auswertung eines Terms zu einem bestimmten Wahrheitswert grundsätzlich verhindert. Es wird lediglich die Überprüfung bestimmter Auswertungsregeln verhindert. Gilt beispielsweise $b \in H_\mathrm{G}$, wobei H_G die aktuellen globalen Hypothesen bezeichnet, so wird b unabhängig von der Polarität durch den Auswertungskalkül zu `true` ausgewertet. Für nicht-boolsche Terme hat die Polarität keine Bedeutung.

- **Die Markierung:** Der Auswertungskalkül muss für einige Anwendungsbedingungen verschiedene Auftreten des gleichen Funktionssymbols f in einem Term t unterscheiden. Eine A-Annotation A enthält daher eine *Markierung* $m \in \mathbb{N}$ mit deren Hilfe diese Unterscheidung möglich ist. Um z.B. in $\texttt{append}(\texttt{x}, \texttt{append}(\texttt{y}, \texttt{z}))$ die beiden Auftreten der Prozedur `append` zu unterscheiden, kann der Term wie folgt markiert werden:

$$\texttt{append}^{\langle\dots,1,\dots\rangle}(\texttt{x}, \texttt{append}^{\langle\dots,2,\dots\rangle}(\texttt{y}, \texttt{z})).$$

- **Das Label:** Um die Auswertung von Prozeduren zu steuern, enthält eine A-Annotation ein so genanntes *Label*. Wir haben bereits oben darauf hingewiesen, dass mit Hilfe der Termsignatur Σ der auszuwertenden Prozeduren einer A-Umgebung lediglich alle Prozeduraufrufe einer Prozedur f gesperrt werden können. Um nur bestimmte Prozeduraufrufe zu sperren, haben wir daraufhin die Kontextmenge C eingeführt, deren Funktionsweise jedoch nicht weiter erklärt wurde. Dies holen wir nun nach. Die Kontextmenge sperrt Prozeduraufrufe, die mit einem bestimmten Label annotiert sind. Hierzu wird zur Auswertung der Argumente t_i eines Prozeduraufrufs $g(t_1, \dots, t_n)$ die Kontextmenge C der A-Umgebung um das Funktionssymbol g erweitert. Ist nun in einem der Argumente t_i ein Prozeduraufruf der Form $f^{\langle\dots,l,\dots\rangle}$ enthalten, wobei l das Label der Annotation bezeichnet und es gilt $l = g$, so ist die Auswertung des Prozeduraufrufs $f^{\langle\dots,l,\dots\rangle}$ gesperrt. Hierbei ist zu beachten, dass Prozeduraufrufe mit $\top$ als Label immer gesperrt sind, da $\top$ nach Definition 6.1 immer in der Kontextmenge enthalten ist.[2]

- **Die A-Umgebung:** Gelegentlich ist es notwendig, einen Teilterm r des auszuwertenden Terms t mit Hilfe einer bestimmten A-Umgebung auszuwerten. A-Annotationen bieten daher die Möglichkeit, Terme mit A-Umgebungen zu annotieren. Ein so annotierter Term wird dann mit der durch die A-Annotation vorgegeben A-Umgebung ausgewertet.

- **Das Search-Limit:** Bei der Überprüfung der Anwendbarkeit von Quantifikationen kann ein unendlicher Suchraum entstehen. A-Annotationen enthalten daher das *Search-Limit* $s \in \mathbb{N}$, welches den Suchraum begrenzt. Ist $s = 0$ so wird keine Quantifikation auf den Term angewendet.

- **Das Unfold-Limit:** Der Auswertungskalkül verwendet zwei Klassen von Regeln zur Auswertung von Prozeduraufrufen. Die Klasse der „*Unfold*"-Regeln wertet Aufrufe einer Prozedur f genau dann aus, wenn das Funktionssymbol f nicht im Ergebnis der Auswertung vorkommt. Hierzu ist es notwendig, den entsprechend instantiierten Prozedurrumpf R_f der Prozedur probeweise symbolisch auszuwerten und anschließend zu überprüfen, ob das Funktionssymbol f im Ergebnis der Auswertung nicht vorkommt. Diese Probeauswertung kann zu einem unendlichen Suchraum führen. Die A-Annotation A enthält daher ein *Unfold-Limit* $u \in \mathbb{N}$, welches den Suchraum begrenzt.

- **Das Instanz-Flag:** Das *Instanz-Flag* bestimmt, für welche `case`-Terme $\texttt{case}(b, \dots, \mathit{cons} : t, \dots)$ die mit den Zweigen assoziierten boolschen Terme

[2] Die Kontextmenge und die Labels sperren keine *trivialen* Prozeduraufrufe. Unter einem trivialen Prozeduraufruf verstehen wir beispielsweise einen Prozeduraufrufe wie $\texttt{plus}(\texttt{1}, \texttt{0})$ wobei `plus` die entsprechende Prozedur aus Abbildung 8.2 bezeichnet. Es ist nicht sinnvoll solche Prozeduraufrufe zu sperren, da sie in der Regel vollständig ausgewertet werden können. Der Prozeduraufrufe $\texttt{plus}(\texttt{1}, \texttt{0})$ beispielsweise kann zu $\texttt{succ}(\texttt{0})$ vereinfacht werden. Welche Prozeduraufrufe als trivial gelten, beschreiben wir ausführlich in Abschnitt 9.1. Für das aktuelle Verständnis ist es jedoch ausreichend anzunehmen, dass Kontext und Labels die Auswertung aller Arten von Prozeduraufrufen sperren können.

?$(cons, b)$ nicht nur als lokale Hypothesen, sondern auch als Instanzhypothesen verwendet werden sollen. Wie dieses Flag gesetzt wird und warum, beschreiben wir im Detail in Kapitel 11.

Damit haben wir die Komponenten unserer A-Annotationen beschrieben und motiviert. Nun definieren wir A-Annotationen formal:

Definition 6.4 (A-Annotationen)**.** Sei P ein terminierendes Programm. Seien weiter

- $m, u, s \in \mathbb{N} \cup \{-\}$,
- $I \in \{\bot, \top\} \cup \{-\}$,
- $p \in \{\oplus, \ominus, \odot\} \cup \{-\}$,
- $l \in \Sigma(P) \cup \{\top, \bot\} \cup \{-\}$ und
- $\mathcal{U} \in \{\mathcal{U}' \mid \mathcal{U}' \text{ ist A-Umgebung von } P\} \cup \{\bot\} \cup \{-\}$.

Eine *Auswertungsannotation* (oder kurz *A-Annotation*) für P ist dann ein Tupel der Form

$$A = \langle p, m, l, \mathcal{U}, u, s, I \rangle.$$

Die Komponenten einer A-Annotationen werden wie folgt bezeichnet:

- p = *Polarität* der A-Annotation.
- m = *Markierung* der A-Annotation.
- l = *Label* der A-Annotation.
- $\mathcal{U}$ = *A-Umgebung* der A-Annotation.
- I = *Instanz-Flag* der A-Annotation.
- u = *Unfold-Limit* der A-Annotation.
- s = *Search-Limit* der A-Annotation.

Der Wert „$-$“ wird als „*don't care value*“ bezeichnet. Für zwei A-Annotationen $A = \langle a_1, \ldots, a_7 \rangle$ und $A' = \langle a'_1, \ldots, a'_7 \rangle$ ist die A-Annotation $A + A'$ wie folgt definiert:

$$A + A' := \langle \overline{a_1}, \ldots, \overline{a_7} \rangle \text{ mit } \overline{a_i} = \begin{cases} a_i & \text{falls } a'_i = -, \\ a'_i & \text{sonst.} \end{cases}$$

Die Menge aller A-Annotationen für P wird mit $\mathcal{A}\mathit{nnot}(P)$ notiert. □

Eine Annotation A bezeichnen wir als *echte* A-Annotation, falls sie keine „*don't care values*“ enthält. Andernfalls bezeichnen wir sie als *unechte* A-Annotation. Wir definieren den Auswertungskalkül über mit echten A-Annotationen annotierten Termen. Mit echten A-Annotationen annotierte Terme wollen wir nachfolgend kurz als *A-Terme* bezeichnen und die Menge aller A-Terme über P mit $\mathcal{A}\mathit{Term}(P)$. Weiter nennen wir mit echten A-Annotationen annotierte Termsubstitution *A-Termsubstitutionen* und notieren die Menge aller A-Termsubstitutionen über P mit $\mathcal{A}\mathit{Subst}(P)$. Der sprachlichen Einfachheit halber bezeichnen wir häufig A-Terme auch einfach als Terme und A-Termsubstitutionen als Termsubstitutionen. Aus dem Zusammenhang ist jeweils klar, ob es sich um Terme oder A-Terme bzw. Termsubstitutionen oder A-Termsubstitutionen handelt. Ist dies nicht der Fall, so sprechen wir

explizit von *nicht-annotierten Termen* und A-Termen bzw. *nicht-annotierten Termsubstitutionen* und A-Termsubstitutionen. Zur Annotierung eines nicht-annotierten Terms t mit einer echten A-Annotation A definieren wir die folgende Funktion:

$$\begin{aligned} \mathtt{a}_A(x) &:= x, \quad (\text{mit } x \in \mathcal{V}(\Omega(P))) \\ \mathtt{a}_A(f(t_1,\ldots,t_n)) &:= f^A(\mathtt{a}_A(t_1),\ldots,\mathtt{a}_A(t_n)), \\ \mathtt{a}_A(\mathtt{let}\ x\mathtt{:=}t\ \mathtt{in}\ r\ \mathtt{end}) &:= \mathtt{let}^A\ x\mathtt{:=}\mathtt{a}_A(t)\ \mathtt{in}\ \mathtt{a}_A(r)\ \mathtt{end}. \end{aligned}$$

Unechte A-Annotation verwenden wir, um einzelne Komponenten der A-Annotationen eines Terms mit neuen Werten zu überschreiben. Wir erweitern dazu die Funktion $\mathtt{a}$ wie folgt:

$$\begin{aligned} \mathtt{a}_{A'}(x) &:= x, \\ \mathtt{a}_{A'}(f^A(t_1,\ldots,t_n)) &:= f^{A+A'}(\mathtt{a}_{A'}(t_1),\ldots,\mathtt{a}_{A'}(t_n)), \\ \mathtt{a}_{A'}(\mathtt{let}^A\ x\mathtt{:=}t\ \mathtt{in}\ r\mathtt{end}) &:= \mathtt{let}^{A+A'}\ x\mathtt{:=}\mathtt{a}_{A'}(t)\ \mathtt{in}\ \mathtt{a}_{A'}(r)\ \mathtt{end}. \end{aligned}$$

6.2 Verwendung der Steuerinformationen

Betrachten wir A-Umgebungen und A-Annotationen genauer, so stellen wir fest, dass verschiedenste Informationen der A-Umgebungen und A-Annotationen zur Steuerung der Auswertung der Prozeduraufrufe eingeführt wurden. Es existieren daher unterschiedliche Abstufungen, um die Auswertung von Prozeduraufrufen zu kontrollieren. Um hierbei nicht den Überblick zu verlieren, stellen wir eine Art Hierarchie der Steuerinformationen auf. Die Hierarchie soll das Verständnis für die nachfolgende Verwendung dieser Steuerinformationen während der Definition des Auswertungskalküls erleichtern. Wir geben die Hierarchie in Form einer einfachen Aufzählung an, wobei wir mit den Steuerinformationen beginnen, die die Auswertung von Prozeduraufrufen am stärksten einschränken:

(1) **Die Termsignatur Σ der auszuwertenden Prozeduren:** Um die Auswertung aller Prozeduraufrufe von Prozeduren für die symbolische Auswertung zu sperren, ist es am einfachsten, die Termsignatur Σ der auszuwertenden Prozeduren der A-Umgebung auf die leere Menge zu setzen. Dadurch werden unabhängig von allen anderen Steuerinformationen keine Prozeduraufrufe mehr ausgewertet. Möchte man lediglich die Auswertung von Prozeduraufrufen für eine ganz bestimmte Prozedur verhindern, so entfernt man nur diese Prozedur aus der Termsignatur Σ.

(2) **Die Menge $\mathcal{R}$ der anzuwendenden Regeln:** Um die Auswertung von Prozeduraufrufen mit Hilfe von bestimmten Regeln des Auswertungskalküls zu verhindern, wird die Nummer der Regel aus der Menge $\mathcal{R}$ der anzuwendenden Regeln entfernt. Weiterhin ist es möglich, mit Hilfe der Menge $\mathcal{R}$ die Auswertung von Prozeduraufrufen insgesamt zu sperren. Hierzu werden lediglich alle Regelnummern aus $\mathcal{R}$ entfernt, deren Regeln Prozeduren auswerten. Im Vergleich zur Verwendung der Termsignatur Σ ist letzteres jedoch recht umständlich. Wir werden daher, falls wir die Auswertung von Prozeduraufrufen komplett sperren möchten, immer die Termsignatur auf die leere Menge setzen und die Menge $\mathcal{R}$ unverändert lassen.

(3) **Die Kontextmenge C und die Labels der Terme:** Um lediglich die Auswertung bestimmter Prozeduraufrufe zu sperren, muss das Label dieser Prozeduraufrufe auf $\top$ gesetzt werden. Soll ein Prozeduraufruf $f(t_1,\ldots,t_n)$ nur innerhalb eines Arguments eines bestimmten anderen Prozeduraufrufs $g(s_1,\ldots,s_n)$ gesperrt werden, so setzt man das Label des Prozeduraufrufs $f(t_1,\ldots,t_n)$ auf die Prozedur g.

(4) **Das Unfold-Limit der Terme:** Um die Auswertung von Prozeduraufrufen mit Hilfe der oben beschriebenen „*Unfold*"-Regeln zu verhindern, wird das Unfold-Limit des entsprechenden Prozeduraufrufs auf 0 gesetzt. Ein solches Zurücksetzen des Unfold-Limits ist meist aus Effizienzgründen notwendig, da die Überprüfung der Anwendungsbedingungen der Regeln relativ aufwendig ist.

Die Anwendung von Quantifikationen für einen Term t können wir auf zweierlei Arten verhindern. Einerseits können wir die Multimenge $\mathcal{Q}$ auf die leere Multimenge setzen und anderseits können wir das Search-Limit aller Teilterme von t auf 0 setzen. Hierbei ist folgendes zu beachten. Für jedes Funktionssymbol $f : \tau_1 \times \ldots \times \tau_n \rightarrow \tau \in \Sigma(P)$ mit nicht-boolschem Ergebnistyp τ ist die folgende Quantifikation in P gültig:

$$\begin{array}{l}\texttt{all}\ \nu^*, \mathtt{x_1}, \mathtt{y_1} : \tau_1, \ldots, \mathtt{x_n}, \mathtt{y_n} : \tau_n \\ \quad \texttt{if}(\mathit{AND}(\mathtt{x_1}{=}\mathtt{y_1}, \ldots, \mathtt{x_n}{=}\mathtt{y_n}), \mathtt{f}(\mathtt{x_1}, \ldots, \mathtt{x_n}){=}\mathtt{f}(\mathtt{y_1}, \ldots, \mathtt{y_n}), \mathtt{true})\end{array} \tag{6.2}$$

Um symbolische Auswertungen bzgl. dieser Quantifikation zu ermöglichen definiert der Auswertungskalkül die „*Functionality*"-Regeln. Da die „*Functionality*"-Regeln ähnliche Suchräume einführen, wie die Regeln zur Anwendung von Quantifikationen, wird die Anwendung der „*Functionality*"-Regeln ebenfalls durch das Search-Limit kontrolliert. Das bedeutet, dass durch Setzen des Search-Limits auf 0 nicht nur die Verwendung der Quantifikationen verhindert wird, sondern auch die Anwendung dieser „*Functionality*"-Regeln. Es ist somit eine Unterschied, ob wir die Multimenge $\mathcal{Q}$ auf die leere Menge setzen oder die Search-Limits der Terme auf 0. Üblicherweise verbieten wir die Verwendung von Quantifikationen, wenn die dadurch eingeführten Suchräume zu ineffizienten Auswertungen führen. Da die „*Functionality*"-Regeln ähnliche Suchräume einführen, müssen wir in diesen Fällen auch die Anwendung der „*Functionality*"-Regeln verbieten. Wir setzen daher in diesen Fällen das Search-Limit auf 0.

Kapitel 7

Der Auswertungskalkül

Nachdem wir nun A-Umgebungen, A-Annotationen und A-Terme definiert haben, können wir den Auswertungskalkül formal einführen. Diese Definition erfolgt in Abschnitt 7.1. Um die Definition der Auswertungsregeln in den Kapiteln 8-13 zu erleichtern, führen wir in den Abschnitten 7.2-7.3 einige Konventionen bzgl. A-Annotationen, A-Umgebungen und Regeldefinitionen ein. Anschließend beschreiben wir in den Abschnitten 7.5-7.7 wichtige Entwurfsprinzipien, die bei den Regeldefinitionen in den Kapiteln 8-13 berücksichtigt wurden. Danach gehen wir in Abschnitt 7.8 kurz auf die im Auswertungskalkül realisierte Gleichheitsbehandlung ein. Das weitere Vorgehen zur Definition der Auswertungregeln beschreiben wir dann in Abschnitt 7.9.

7.1 Definition des Auswertungskalküls

Definition 7.1 (Auswertungskalkül)**.** Sei P ein terminierendes Programm und $\mathcal{U}$ eine A-Umgebung für P. Der *Auswertungskalkül* (oder kurz *A-Kalkül*) ist dann wie folgt definiert:

(1) **Sprache:** Die A-Terme über P.

(2) **Inferenzregeln:** Die in den Kapiteln 8-13 definierten Regeln der Form

Regelnummer. Regelname
$$\frac{t}{r}, \quad \text{falls } \Phi(t, r).$$

(3) **Deduktion:** Wir schreiben $\mathcal{U} \vdash t \to t'$ falls t' aus t mittels einer Anwendung einer Regel der Form

$$\frac{t}{t'}$$

bezüglich $\mathcal{U}$ entsteht, wobei die Regeln in der Reihenfolge der Regelnummer überprüft werden. Für eine Folge $\langle t_i \rangle_{i \in \{1,\ldots,n\}}$ von annotierten Termen mit $\mathcal{U} \vdash t_i \to t_{i+1}$ für alle $i = 1, \ldots, n-1$ schreiben wir $\mathcal{U} \vdash t_1 \to \ldots \to t_n$ bzw. $\mathcal{U} \vdash t_1 \to^* t_n$. Existiert kein t' mit $\mathcal{U} \vdash t \to t'$, so bezeichnen wir t als $\mathcal{U}$-ausgewertet. Gilt $\mathcal{U} \vdash t \to^* t'$ und ist t' $\mathcal{U}$-ausgewertet, so nennen wir t' eine $\mathcal{U}$-Normalform von t und wir schreiben $\mathcal{U} \vdash t \to^! t'$. Durch die feste Reihenfolge der Regelanwendungen sowie die Determiniertheit jeder Regelanwendung

existiert für jeden Term t höchstens eine $\mathcal{U}$-Normalform. Wir bezeichnen diese $\mathcal{U}$-Normalform mit $t\downarrow_{\mathcal{U}}$.

□

7.2 Konventionen für A-Umgebungen

In der Anwendungsbedingung $\Phi(t,r)$ einer Regel bezeichnet $\mathcal{U}$ immer die aktuelle A-Umgebung unter der t zu r ausgewertet werden soll. Die einzelnen Komponenten von $\mathcal{U}$ werden durch folgende Bezeichner referenziert:

$$\langle P, \Sigma, \Sigma_{\mathrm{C}}, \Sigma_{\mathrm{A}}, \mathcal{Q}, H_{\mathrm{L}}, H_{\mathrm{G}}, H_{\mathrm{I}}, C, \mathcal{R}, \delta\rangle.$$

Falls wir in einer Regel eine neue A-Umgebung $\mathcal{U}'$ definieren müssen, so tun wir dies mit Hilfe einer Tabelle:

	Prog.	Proz.	Komm.	Ass.	Quant.	$\mathrm{Hyp}_{\mathrm{L}}$	$\mathrm{Hyp}_{\mathrm{G}}$	$\mathrm{Hyp}_{\mathrm{I}}$	Ktx.	Reg.	Bind.
$\mathcal{U}'$:	P'	Σ'	Σ'_{C}	Σ'_{A}	$\mathcal{Q}'$	H'_{L}	H'_{G}	H'_{I}	C'	$\mathcal{R}'$	δ'

Die einzelnen Komponenten der A-Umgebung $\mathcal{U}'$ werden durch die entsprechende Spalte definiert, wobei die Überschrift der Spalte die Komponente eindeutig identifiziert. Um die Tabellen klein und übersichtlich zu halten, müssen nicht alle Komponenten der neuen A-Umgebung $\mathcal{U}'$ explizit angegeben werden. Komponenten, die in der Tabelle nicht aufgeführt sind, werden auf den Wert der entsprechenden Komponente der aktuellen A-Umgebung $\mathcal{U}$ gesetzt. So definiert z.B. die Tabelle

	$\mathrm{Hyp}_{\mathrm{L}}$
$\mathcal{U}'$:	H'_{L}

eine A-Umgebung $\mathcal{U}'$ mit

$$\mathcal{U}' = \langle P, \Sigma, \Sigma_{\mathrm{C}}, \Sigma_{\mathrm{A}}, \mathcal{Q}, H'_{\mathrm{L}}, H_{\mathrm{G}}, H_{\mathrm{I}}, C, \mathcal{R}, \delta\rangle.$$

Definieren wir eine neuen A-Umgebung $\mathcal{U}''$ auf Basis einer anderen A-Umgebung $\mathcal{U}'$, so geben wir den Namen der A-Umgebung $\mathcal{U}'$ in der Tabelle explizit an. Beispielsweise notieren wir eine A-Umgebung $\mathcal{U}''$, die aus einer A-Umgebung $\mathcal{U}'$ durch Zurücksetzen der Quantifikationen auf die leere Multimenge hervorgeht wie folgt:

$\mathcal{U}'$:	Quant.
$\mathcal{U}''$:	$\emptyset$

Gelegentlich ist es notwendig einen Term auf Basis von aussagenlogischen Umformungen zu vereinfachen. Hierzu werten wir den Term mit der fest definierten A-Umgebung $\mathcal{U}_0$ aus:

	Prog.	Proz.	Komm.	Ass.	Quant.	$\mathrm{Hyp}_{\mathrm{L}}$	$\mathrm{Hyp}_{\mathrm{G}}$	$\mathrm{Hyp}_{\mathrm{I}}$	Ktx.	Reg.	Bind.
$\mathcal{U}_0$:	$\langle\rangle$	$\emptyset$	$\emptyset$	$\emptyset$	$\emptyset$	$\emptyset$	$\emptyset$	$\emptyset$	$\emptyset$	$\mathcal{R}_0$	$\emptyset$

Die Menge der zulässigen Regeln ist hierbei wie folgt gegeben:

$$\mathcal{R}_0 := \{1, \ldots, R_{\max}\} \setminus \{49, 50\}^1$$

Wir werden häufig zur Illustration und Motivation von Regeln Beispielauswertungen präsentieren. Um die Darstellung dieser Beispielauswertungen zu vereinfachen, treffen wir für die hierbei verwendeten A-Umgebungen folgende Vereinbarung:

[1] Ein Verzeichnis aller Regeln geordnet nach Nummern befindet sich auf Seite 285. Die entfernten Regeln 49 und 50 werden grundsätzlich nur während der Auswertung der Subgoals von Quantifikationen angewendet. Siehe hierzu Kapitel 11.

Das Programm der A-Umgebung beinhaltet, sofern nichts anderes gesagt wird, immer alle notwendigen Typoperator- und Prozedurdefinitionen. Die Termsignatur Σ der auszuwertenden Prozeduren enthält alle durch das Programm definierten Prozeduren. Falls wir die Auswertung eines Goal-Terms einer Sequenz betrachten, so ist die globale Hypothesenmenge durch die Hypothesenmenge der Sequenz gegeben. Andernfalls ist die globale Hypothesenmenge leer. Weiter sind alle anderen Komponenten der A-Umgebung, sofern nichts anderes gesagt wird, durch die leere Menge bzw. leere Multimenge gegeben. Betrachten wir beispielsweise nachfolgend eine Auswertung wie

$$\mathcal{U} \vdash \texttt{pred}(\texttt{succ}(\texttt{x})) \to \texttt{x},$$

ohne die A-Umgebung $\mathcal{U}$ näher zu beschreiben, so ist diese A-Umgebung in diesem Fall durch folgende Tabelle definiert:

	Prog.	Proz.	Komm.	Ass.	Idem.	Quant.	Hyp_L	Hyp_G	Hyp_I	Ktx.	Reg.	Bind.
$\mathcal{U}$:	$\langle\rangle$	$\emptyset$	$\emptyset$	$\emptyset$	$\emptyset$	$\emptyset$	$\emptyset$	$\emptyset$	$\emptyset$	$\emptyset$	$\emptyset$	$\emptyset$

7.3 Konventionen für A-Annotationen

Die A-Annotationen eines A-Terms erschweren die Lesbarkeit des A-Terms enorm. Grundsätzlich notieren wir daher nur die A-Annotationen, die für den aktuellen Zusammenhang relevant sind. Alle anderen A-Annotationen werden nicht aufgeführt und sind daher als *beliebig aber fest* anzusehen. Insbesondere während der Motivation von Regeln notieren wir häufig gar keine A-Annotationen eines A-Terms. Die A-Terme werden dann durch die entsprechenden nicht-annotierten Terme dargestellt. Es sei an dieser Stelle aber explizit darauf hingewiesen, dass es sich dabei trotzdem um A-Terme handelt. Da die Annotationen für die Konstruktorgrundterme `true` und `false` nie verwendet werden, notieren wir diese Terme grundsätzlich ohne Annotationen.

Allgemein notieren wir die aufgeführten A-Annotationen eines A-Terms grau unterlegt. Damit werden die A-Annotationen optisch von der eigentlichen Termstruktur getrennt, wodurch die Termstruktur klarer hervortritt. Weiter notieren wir die Komponenten der A-Annotationen meistens nicht im Term direkt, sondern – ähnlich wie bei A-Umgebungen – mit Hilfe von Tabellen. Den A-Term

$$\texttt{f}^{\,\langle p,m,l,u,s,I\rangle}(\texttt{g}^{\,\langle p,m,l,u',s,I\rangle}(\texttt{x}))$$

notieren wir dann mit $\texttt{f}^{\,A}(\texttt{g}^{\,B}(\texttt{x}))$ und der folgenden Tabelle:

	Pol.	Mark.	Lab.	U-Lim.	S-Lim.	Inst.
A:	p	m	l	u	s	I
B:	p	m	l	u'	s	I

Häufig leiten wir eine A-Annotation B aus einer A-Annotation A ab. Das bedeutet, die A-Annotation B stimmt mit der A-Annotation A bis auf einige wenige Komponenten überein. Wir treffen daher folgende Vereinbarung: In einer Tabelle zur Definition der A-Annotation B müssen nur die Komponenten aufgeführt werden, in denen sich die A-Annotation B von der A-Annotation A unterscheiden. Die A-Annotation A, aus der wir die A-Annotation B ableiten, notieren wir dann in der linken oberen Ecke der Tabelle. Die Tabelle

A:	S-Lim.
B:	0

definiert beispielsweise die A-Annotation B als eine A-Annotation, die bis auf das Search-Limit mit der A-Annotation A übereinstimmt. Das Search-Limit der A-Annotation B wird durch die Tabelle auf 0 gesetzt.

Einige A-Annotationen treten nachfolgend immer wieder auf. Um diese A-Annotationen einfach verwenden zu können, führen wir für diese Annotationen feste Namen ein:

	Pol.	Mark.	Lab.	U-Lim.	S-Lim.	Inst.
A_{std}:	$\odot$	0	$\bot$	0	0	$\bot$
$A_{\max}$:	$\odot$	0	$\bot$	$u_{\max}$	$s_{\max}$	$\top$
A_{top}:	$\odot$	0	$\top$	0	0	$\bot$
A_0:	–	–	–	0	0	–

Hierbei bezeichnen $u_{\max}$ und $s_{\max}$ die von einer ✓eriFun-Installation fest vorgegebenen, maximalen Werte für das Unfold-Limit und das Search-Limit. Der Goal-Term der initialen Sequenz des Beweisbaums eines Lemmas wird mit $A_{\max}$ annotiert.

Für endliche Listen annotierter Terme des Typs `bool` legen wir die Annotationen von Konjunktion und Disjunktion wie folgt fest:

$$\begin{aligned}
AND(\langle\rangle) &:= \texttt{true}, \\
AND(\langle l\rangle) &:= l, \\
AND(\langle l_1, l_2 \ldots, l_n\rangle) &:= \texttt{if}^{\,A_{\text{std}}}(l_1, AND(\langle l_2, \ldots, l_n\rangle), \texttt{false}),
\end{aligned}$$

$$\begin{aligned}
OR(\langle\rangle) &:= \texttt{false}, \\
OR(\langle l\rangle) &:= l, \\
OR(\langle l_1, l_2 \ldots, l_n\rangle) &:= \texttt{if}^{\,A_{\text{std}}}(l_1, \texttt{true}, OR(\langle l_2, \ldots, l_n\rangle)).
\end{aligned}$$

Für $AND(\langle l_1, \ldots, l_n\rangle)$ bzw. $OR(\langle l_1, \ldots, l_n\rangle)$ schreiben wir – ähnlich wie für Konjunktionen und Disjunktionen nicht-annotierter Terme – auch kurz $AND(l_1, \ldots, l_n)$ bzw. $OR(l_1, \ldots, l_n)$. Konjunktionen und Disjunktionen endlicher Mengen annotierter Terme des Typs `bool` sind analog zu Konjunktionen und Disjunktionen endlicher Mengen nicht-annotierter Terme definiert (siehe Abschnitt 4.1).

7.4 Konventionen für Regeldefinitionen

Um die Entwicklung der Regeln in den Kapiteln 8-13 aufzuzeigen, definieren wir gelegentlich vorläufige Regeln. Diese notieren wir ohne Angabe eines Namens und einer Nummer direkt im Text, z.B.

$$\frac{t}{r}, \qquad \text{falls } \Phi(t, r).$$

Weiter ist es häufig notwendig, Auswertungen zu betrachten, die nicht alle Regeln des Auswertungskalküls verwenden. Wir führen daher folgende Konvention ein: Sofern nichts anderes gesagt wird, verwenden alle Auswertungen ausschließlich die Regeln, die bis zu dem jeweiligen Zeitpunkt bereits definiert wurden.

7.5 Termvergleiche und Mengenoperatoren

In den Auswertungsregeln ist es häufig notwendig Terme zu vergleichen. Wir verwenden hierzu grundsätzlich die Relation $\simeq_{\mathcal{U}}$.

Definition 7.2 ($\simeq_{\mathcal{U}}$)**.** Sei $\mathcal{U}$ eine A-Umgebung. Die Relationen $\simeq_{\mathcal{U}} \subseteq \mathcal{T}erm(P) \times \mathcal{T}erm(P)$ ist definiert durch

$$t \simeq_{\mathcal{U}} r \quad \text{gdw.} \quad \delta(t) =^*_{\mathcal{U}} \delta(r),$$

wobei die Relation $=_{\mathcal{U}}$ die kleinste Kongruenzrelation auf $\mathcal{T}erm(P)$ bezeichnet, für die gilt $f(t,r) =_{\mathcal{U}} f(r,t)$ für alle $f \in \Sigma_{\mathrm{C}} \cup \{\mathtt{=}\}$. □

Dadurch die Verwendung der Relation $\simeq_{\mathcal{U}}$ gewährleisten wir nicht nur die Berücksichtigung der Symmetrie der Gleichheit, sondern auch die Berücksichtigung der kommutativen Funktionssymbole und der aktuellen Variablenbindungen. Die Festlegung wirkt sich auch auf Mengenoperatoren wie $\in, \cup, \cap, \ldots$ aus. Anstatt dieser Mengenoperatoren verwenden wir im Auswertungskalkül die Mengenoperatoren $\in_{\simeq_{\mathcal{U}}}, \cup_{\simeq_{\mathcal{U}}}, \cap_{\simeq_{\mathcal{U}}}, \ldots$.[2]

7.6 Nachweis der Gültigkeit von Literalen

Häufig ist es in einer Auswertungsregel notwendig die Gültigkeit eines Literals l zu überprüfen. Hierzu verwenden wir eines der beiden folgenden Verfahren:

(1) Wir überprüfen, ob die Gültigkeit des Literals l aus den lokalen und globalen Hypothesen folgt. Hierzu vervollständigen wir zunächst mit Hilfe der Funktion $complete_P$ aus Abschnitt 4.2 die Menge $H_{\mathrm{L}} \cup H_{\mathrm{G}}$ um alle Strukturtests, deren Gültigkeit aus $H_{\mathrm{L}} \cup H_{\mathrm{G}}$ mit Hilfe der Vollständigkeit und Eindeutigkeit der Strukturtests folgt (vergleiche Abschnitt 3.5). Anschließend prüfen wir, ob das Literal in der vervollständigten Menge enthalten ist. Ist dies der Fall, so haben wir die Gültigkeit des Literals bewiesen. Insgesamt prüfen wir also die Gültigkeit des Literals l wie folgt:

$$l \in_{\simeq_{\mathcal{U}}} complete_P(H_{\mathrm{L}} \cup H_{\mathrm{G}}).$$

(2) Wir werten das Literal l symbolisch aus. Hierzu verwenden wir eine an das aktuelle Beweisproblem angepasste A-Umgebung $\mathcal{U}'$. Das bedeutet, wir überprüfen die Gültigkeit des Literals l durch folgende Anwendungsbedingung:

$$\mathcal{U}' \vdash l \rightarrow^{!} \mathtt{true}.$$

7.7 Transparenz bei Auswertung von Termen

Auswertungsregeln, die zur Überprüfung ihrer Anwendungsbedingung Terme mit Hilfe des Auswertungskalküls symbolisch auswerten, bezeichnen wir als *rekursive Auswertungsregeln*. Beispiele für solche rekursiven Auswertungsregeln sind die in Kapitel 11 definierten „*Assumption*"-Regeln. Bei der Definition rekursiver Auswertungsregeln ist darauf zu achten, dass die Anwendung der Regel transparent, d.h. nachvollziehbar ist. Der Ergebnisterme r einer rekursiven Auswertungsregel der Form

$$\frac{t}{r}, \quad \text{falls } \Phi(t,r)$$

definieren wir daher grundsätzlich so, dass die symbolischen Auswertungen der Anwendungsbedingung $\Phi(t,r)$ in der symbolischen Auswertung von r erneut berechnet werden. So werden beispielsweise die Teilterme eines Terms grundsätzlich schrittweise ausgewertet. Wir definieren hierzu eine Reihe von Auswertungsregeln der Form

$$\frac{f(\ldots,t,\ldots)}{f(\ldots,t',\ldots)}, \quad \text{falls } \mathcal{U} \vdash t \rightarrow t' \text{ und } \ldots$$

[2] Für eine Definition der Mengenoperatoren $\in_{\approx}, \cup_{\approx}, \cap_{\approx}, \ldots$ für eine beliebige Äquivalenzrelation $\approx$ verweisen wir auf den Anhang A.

Eine alternative Definition hierzu wäre, die Argumente direkt durch ihre $\mathcal{U}$-Normalformen zu ersetzen:

$$\frac{f(\ldots,t,\ldots)}{f(\ldots,t\downarrow_{\mathcal{U}},\ldots)}, \quad \text{falls } \mathcal{U}\vdash t\to t' \text{ und } \ldots \tag{7.1}$$

Diese Definition hätte den Nachteil, dass symbolische Auswertungen nur schwer lesbar wären. So würde sich für den vollständig definierten A-Kalkül die Auswertung von

$$\texttt{if}(\texttt{?0}(\texttt{plus}(\texttt{succ}(\texttt{x}),\texttt{0})),\texttt{false},\texttt{true})$$

mit der alternativen Definition wie folgt darstellen:

$$\begin{array}{lll} \mathcal{U} \vdash & \texttt{if}(\underline{\texttt{?0}(\texttt{plus}(\texttt{succ}(\texttt{x}),\texttt{0}))},\texttt{false},\texttt{true}) & \to \\ & \underline{\texttt{if}(\texttt{false},\texttt{false},\texttt{true})} & \to \\ & \texttt{true}\,. & \end{array}$$

Wie der Teilterm $\texttt{?0}(\texttt{plus}(\texttt{succ}(\texttt{x}),\texttt{0}))$ zu `false` ausgewertet wurde, wäre nicht ersichtlich. Mit Hilfe unserer definierten Auswertungsregeln erhalten wir die folgende Auswertung:

$$\begin{array}{lll} \mathcal{U} \vdash & \texttt{if}(\texttt{?0}(\texttt{plus}(\texttt{succ}(\texttt{x}),\texttt{0})),\texttt{false},\texttt{true}) & \to \\ & \texttt{if}(\texttt{?0}(\texttt{if}(\underline{\texttt{?0}(\texttt{succ}(\texttt{x}))},\texttt{0},\texttt{succ}(\texttt{plus}(\ldots)))),\texttt{false},\texttt{true}) & \to \\ & \texttt{if}(\texttt{?0}(\underline{\texttt{if}(\texttt{false},\texttt{0},\texttt{succ}(\texttt{plus}(\ldots)))}),\texttt{false},\texttt{true}) & \to \\ & \texttt{if}(\underline{\texttt{?0}(\texttt{succ}(\texttt{plus}(\ldots)))},\texttt{false},\texttt{true}) & \to \\ & \underline{\texttt{if}(\texttt{false},\texttt{false},\texttt{true})} & \to \\ & \texttt{true}\,. & \end{array}$$

7.8 Gleichheitsbehandlung

Der Auswertungskalkül definiert keine spezielle Komponente zur Gleichheitsbehandlung. Vielmehr werden die Eigenschaften der Gleichheit durch verschiedene Teile des Auswertungskalküls implementiert. Durch diese Verteilung ist eine zusammenhängende Betrachtung der Gleichheitsbehandlung während der Definition der Auswertungsregeln schwierig. Wir wollen daher an dieser Stelle einen kurzen Überblick über die im Auswertungskalkül realisierte Gleichheitsbehandlung geben. Hierzu beschreiben wir, wie die einzelnen Eigenschaften der Gleichheit im Auswertungskalkül implementiert sind. Die Gleichheit ist definiert als die kleinste Kongruenzrelation auf $\mathcal{T}\!erm(P)$, d.h. es gilt:[3]

$$\texttt{x=x} \equiv_P \texttt{true} \qquad (\textit{Reflexivität})$$

$$\texttt{if}(\texttt{x=y},\texttt{y=x},\texttt{true}) \equiv_P \texttt{true} \qquad (\textit{Symmetrie})$$

$$\texttt{if}(\texttt{x=y},\texttt{if}(\texttt{y=z},\texttt{x=z},\texttt{true}),\texttt{true}) \equiv_P \texttt{true} \qquad (\textit{Transitivität})$$

$$\texttt{if}(\texttt{y=z}, f(\texttt{x}_1,\ldots,\texttt{y},\ldots,\texttt{x}_n)\texttt{=}f(\texttt{x}_1,\ldots,\texttt{z},\ldots,\texttt{x}_n),\texttt{true}) \equiv_P \texttt{true} \qquad (\textit{Kongruenz})$$

Diese Eigenschaften sind im Auswertungskalkül wie folgt realisiert:

[3] Die Kongruenz ist für jedes Funktionssymbol f des Programms P gültig.

- **Reflexivität:** Für die Reflexivität ist im Auswertungskalkül eine eigene Regel definiert. Diese Regel vergleicht die linke und die rechte Seite einer Gleichung mittels der Relation $\simeq_{\mathcal{U}}$ und ersetzt gegebenenfalls die Gleichung durch `true`.
- **Symmetrie:** Die Symmetrie der Gleichheit wird im Auswertungskalkül dadurch berücksichtigt, dass in allen Auswertungsregeln Terme bzgl. der Relation $\simeq_{\mathcal{U}}$ verglichen werden.
- **Transitivität:** Der Auswertungskalkül definiert keine explizite Behandlung für die Transitivität der Gleichheit. Statt dessen ist in ✓`eriFun` ein Transitivitätsaxiom für die Gleichheit vordefiniert. Dieses Axiom wird für die Auswertungen genauso behandelt, wie die verifizierten Lemmata eines Programms.
- **Kongruenz:** Die Kongruenzeigenschaft der Gleichheit wird durch verschiedene Auswertungsregeln realisiert, etwa durch die „*Assumption replacement*"-Regel zur bedingten Termersetzung. Weiter existieren Regeln um verschiedene Varianten der Quantifikation (6.2) zu implementieren.

7.9 Weiteres Vorgehen

Zur Definition der Auswertungsregeln gehen wir nachfolgend wie folgt vor. Wir definieren in Kapitel 8 zunächst die Basisregeln des Auswertungskalküls. Hierbei handelt es sich um Regeln zur Auswertung von `case`-Termen, `let`-Termen, Gleichungen, etc. Anschließend führen wir in Kapitel 9 die „*Execute*"-Regeln zur Auswertung von Prozeduraufrufen ein. Bei den „*Execute*"-Regeln handelt es sich um die maßgeblichen Regeln des A-Kalküls zur Auswertung von Prozeduraufrufen während eines Induktionsbeweises. Neben den „*Execute*"- Regeln existieren noch die „*Unfold*"-Regeln zur Auswertung von Prozeduraufrufen. Diese Regeln werten Prozeduraufrufe unter Berücksichtigung des Termkontexts aus. Wir definieren die „*Unfold*"-Regeln in Kapitel 10. Danach führen wir in den Kapiteln 11 und 12 die „*Assumption*" und „*Replacement*"-Regeln ein, um einen Term mit Hilfe von Induktionshypothesen und Lemmata zu vereinfachen. Die Effizienz und Mächtigkeit des A-Kalküls hängt entscheidend von der Definition der „*Assumption*" und „*Replacement*"-Regeln ab. Schließlich führen wir in Abschnitt 13 die „*Functionality*"-Regeln ein, um Gleichungen auf Basis verschiedene Varianten der Quantifikation (6.2) zu vereinfachen.

Kapitel 8

Die Basisregeln

Wir definieren in diesem Kapitel die Basisregeln des Auswertungskalküls, d.h. wir definieren Regeln zur Auswertung von `case`-Termen, `let`-Termen, Gleichungen, etc. Zur Definition der Basisregeln gehen wir wie folgt vor. In den Abschnitten 8.1-8.2 definieren wir Regeln zur Anwendung globaler und lokaler Hypothesen sowie zur $\mathcal{L}$-Normalisierung von Termen. Anschließend führen wir in den Abschnitten 8.3-8.6 Regeln zur Auswertung von `case`-Termen, `let`-Termen, Gleichungen, Strukturprädikaten und Selektoren ein. Danach definieren wir in Abschnitt 8.7 eine Regel zur Auswertung der Argumente von Funktionsaufrufen. In Abschnitt 8.8 führen wir dann eine Regel ein, um die Atome von boolschen `if`-Termen in geeigneter Weise anzuordnen. Schließlich definieren wir in Abschnitt 8.9 technische Regeln, um das Auswertungsverhalten des Kalküls zu steuern.

In den meisten der Abschnitte 8.1-8.9 geben wir am Anfang eine Liste der im jeweiligen Abschnitt definierten Regeln an. Hierbei nennen wir jeweils den Namen und die Nummer der Regel. Die Reihenfolge der Regeln in den Listen entspricht der Anwendungsreihenfolge der Regeln. In Abbildung 8.1 ist eine vollständige Liste aller in diesem Kapitel definierten Regeln dargestellt. Hierbei wurden die Regeln der Übersichtlichkeit halber zu Gruppen, die bestimmte Aufgaben kennzeichnen, zusammengefasst.

8.1 Regeln für Hypothesen

Wir definieren in diese Abschnitt Regeln zur Verwendung der lokalen und globalen Hypothesen. Ist der auszuwertende Term a ein Atom und folgt die Gültigkeit von a bzw. die Gültigkeit der Negation $\neg a$ aus den lokalen und globalen Hypothesen, so können wir a zu `true` bzw. `false` auswerten. Die entsprechenden Regeldefinitionen lauten wie folgt:

6. Affirmative hypothesis

$$\frac{a}{\texttt{true}},$$

falls $\mathsf{e}(a) \in \mathcal{A}t(P)$ und $\mathsf{e}(a) \in_{\simeq_\mathcal{U}} complete_P(H_L \cup H_G)$.

1.	*Set e-environment*	Technische Regeln
2.	*Reset e-environment*	
3.	*Evaluate e-environment argument*	
4.	*Evaluate e-environment let argument*	
5.	*Evaluate e-environment let body*	
6.	*Affirmative Hypothesis*	Hypothesen und `case`-Terme und $\mathcal{L}$-Normalisierung
7.	*Negative Hypothesis*	
8.	*Keep branch*	
9.	*Skip negation*	
10.	*Skip alternatives*	
11.	*Merge branches*	
12.	*Skip condition*	
13.	*Evaluate condition*	
14.	*Distribute condition*	
15.	*Distribute let body condition*	
16.	*Replace condition*	
17.	*Skip branch*	
18.	*Evaluate branch*	
19.	*Skip let*	`let`-Terme und $\mathcal{L}$-Normalisierung
20.	*Evaluate let assignment*	
21.	*Replace let assignment*	
22.	*Move let assignment*	
23.	*Distribute let assignment*	
24.	*Split let assignment*	
25.	*Evaluate let body*	
26.	*Distribute let body*	
27.	*Reflexivity*	Gleichungen und Strukutrprädiktae und Selektoren
28.	*Constructor uniqueness*	
29.	*Constructor injectivity*	
30.	*Affirmative structure test*	
31.	*Negative structure test*	
32.	*Appropriate selector*	
33.	*Structure equation to structure test*	
34.	*Replace structure predicate argument*	
36.	*Evaluate argument*	Argumente und $\mathcal{L}$-Normalisierung und Gleichungen
37.	*Distribute argument*	
38.	*Distribute let argument*	
39.	*Replace left equality argument*	
40.	*Replace right equality argument*	
79.	*Move then-part atom*	Regeln zum Anordnen der Atome
80.	*Move else-part atom*	
81.	*Commute then-part atom*	
82.	*Commute else-part atom*	

Abbildung 8.1: Auswertungsreihenfolge der in Kapitel 8 definierten Regeln.

7. Negative hypothesis

$$\frac{a}{\texttt{false}},$$

falls $\mathsf{e}(a) \in \mathcal{A}t(P)$ und $\neg\mathsf{e}(a) \in_{\simeq_\mathcal{U}} complete_P(H_L \cup H_G)$.

8.2 Regeln zur $\mathcal{L}$-Normalisierung

In Abschnitt 3.7 haben wir den Begriff der $\mathcal{L}$-normalisierten Terme eingeführt. Für die Teilterme dieser Terme gilt, dass alle Bedingungen von `case`-Termen und alle Variablenbindungen von `let`-Termen `case`- und `let`-frei sind. $\mathcal{L}$-normalisierte Terme sind somit einfacher zu lesen und zu verstehen als äquivalente nicht-$\mathcal{L}$-normalisierte Terme. Für die symbolische Auswertung ist die Betrachtung $\mathcal{L}$-normalisierter Terme vorteilhaft, da sich so die Anzahl der zu definierenden Regeln verringert. Wir führen daher die folgenden Regeln zu $\mathcal{L}$-Normalisierung von Termen ein:

$$\begin{array}{ll}
\vdots & \vdots \\
14. & \textit{Distribute condition} \\
15. & \textit{Distribute let body condition} \\
\vdots & \vdots \\
22. & \textit{Move let assignment} \\
23. & \textit{Distribute let assignment} \\
\vdots & \vdots \\
37. & \textit{Distribute argument} \\
38. & \textit{Distribute let argument} \\
\vdots & \vdots
\end{array}$$

Die Anwendungsreihenfolge dieser Regeln ist so definiert, dass ein Term der Form

$$f(\ldots, \underbrace{\texttt{case}(\ldots)}_{t}, \ldots) \quad \text{bzw.} \quad f(\ldots, \underbrace{\texttt{let ... in ... end}}_{t'}, \ldots)$$

erst dann $\mathcal{L}$-normalisiert wird, wenn t bzw. t' durch keine andere Regel des A-Kalküls ausgewertet werden kann. Die Regeldefinitionen lauten im einzelnen wie folgt:

14. Distribute condition

$$\frac{\texttt{case}^{\,A}(\texttt{case}^{\,B}(b, cons'_1 : r_1, \ldots, cons'_m : r_m), cons_1 : t_1, \ldots, cons_n : t_n)}{\begin{array}{l}\texttt{case}^{\,B}(b, cons'_1 : \texttt{case}^{\,A}(r_1, cons_1 : t_1, \ldots, cons_n : t_n), \\ \qquad \ldots, \\ \qquad cons'_m : \texttt{case}^{\,A}(r_m, cons_1 : t_1, \ldots, cons_n : t_n))\end{array}}$$

15. Distribute let body condition

$$\frac{\texttt{case}^{\,A}(\texttt{let}^{\,B}\ x\texttt{:=}r\ \texttt{in}\ b\ \texttt{end}, cons_1 : r_1, \ldots, cons_l : r_l)}{\texttt{let}^{\,B}\ x'\texttt{:=}r\ \texttt{in case}^{\,A}(b[x/x'], cons_1 : r_1, \ldots, cons_l : r_l)\ \texttt{end}}$$

mit $x' = Var(fv(r_1) \cup \ldots, fv(r_n))$.

22. Move let assignment

$$\frac{\texttt{let}^{\,A}\ x\texttt{:=}(\texttt{let}^{\,B}\ y\texttt{:=}t\ \texttt{in}\ r\ \texttt{end})\ \texttt{in}\ r'\ \texttt{end}}{\texttt{let}^{\,B}\ y'\texttt{:=}t\ \texttt{in}\ (\texttt{let}^{\,A}\ x\texttt{:=}r[y/y']\ \texttt{in}\ r'\ \texttt{end})\ \texttt{end}}$$

mit $y' = Var(fv(r) \cup fv(r'))$.

23. Distribute let assignment

$$\frac{\texttt{let}^{\,A}\ x\texttt{:=case}^{\,B}(b, cons_1 : t_1, \ldots, cons_l : t_l)\ \texttt{in}\ r\ \texttt{end}}{\begin{array}{l}\texttt{case}^{\,B}(b, cons_1 : (\texttt{let}^{\,A}\ x\texttt{:=}t_1\ \texttt{in}\ r\ \texttt{end}),\\ \qquad \ldots,\\ \qquad cons_l : (\texttt{let}^{\,A}\ x\texttt{:=}t_l\ \texttt{in}\ r\ \texttt{end}))\end{array}}$$

37. Distribute argument

$$\frac{f^{\,A}(t_1, \ldots, t_{i-1}, \texttt{case}^{\,B}(b, cons_1 : r_1, \ldots, cons_l : r_l), t_{i+11}, \ldots, t_n)}{\begin{array}{l}\texttt{case}^{\,B}(b, cons_1 : f^{\,A}(t_1, \ldots, t_{i-1}, r_1, t_{i+1}, \ldots, t_n),\\ \qquad \ldots,\\ \qquad cons_l : f^{\,A}(t_1, \ldots, t_{i-1}, r_l, t_{i+1}, \ldots, t_n))\end{array}},$$

falls $f \notin \Sigma^{\text{case}}(P)$ und für alle $j \in \{1, \ldots, i-1\}$ gilt t_j ist kein `case`-Term.

38. Distribute let argument

$$\frac{f^{\,A}(t_1, \ldots, t_{i-1}, (\texttt{let}^{\,B}\ x\texttt{:=}r\ \texttt{in}\ t), t_{i+1}, \ldots, t_n)}{\texttt{let}^{\,B}\ x'\texttt{:=}r\ \texttt{in}\ f^{\,A}(t_1, \ldots, t_{i-1}, t[x/x'], t_{i+1}, \ldots, t_n)\ \texttt{end}},$$

falls $f \notin \Sigma^{\text{case}}(P)$, $x' = Var(fv(f(t_1, \ldots, t, \ldots, t_n)))$ und für alle $j \in \{1, \ldots, i-1\}$ gilt t_j ist kein `let`-Term.

Zur Illustration der Regeln betrachten wir ein Beispiel:

Beispiel 8.1. Sei das Programm P durch $\langle D_{\texttt{plus}}, D_{\texttt{dbl}}\rangle$ gegeben, wobei $D_{\texttt{plus}}$ und $D_{\texttt{dbl}}$ die entsprechenden Prozedurdefinitionen aus Abbildung 8.2 bezeichnen. Betrachten wir nun den folgenden Term:

$$\texttt{if(let z:=plus(x,y) in dbl(z) end=0,x,y)=0}.$$

Die symbolische Auswertung des Terms sieht wie folgt aus:[1]

[1] Wir haben hierbei angenommen, dass zur Auswertung von Gleichungen die Argumente der Gleichungen ausgewertet werden dürfen.

$D_{\texttt{plus}} = \texttt{function plus(x : nat, y : nat) : nat} \Leftarrow$
```
if(?0(x),
   y,
   succ(plus(pred(x), y)))
```

$D_{\texttt{dbl}} = \texttt{function dbl(x : nat) : nat} \Leftarrow$
```
if(?0(x),
   0,
   succ(succ(dbl(pred(x)))))
```

$D_{\texttt{half}} = \texttt{function half(x : nat) : nat} \Leftarrow$
```
if(?0(x),
   0,
   if(?0(pred(x)),
      0,
      succ(half(pred(pred(x))))))
```

$D_{\texttt{even}} = \texttt{function even(x : nat) : nat} \Leftarrow$
```
if(?0(x),
   true,
   if(?0(pred(x)),
      false,
      even(pred(pred(x)))))
```

$D_{\texttt{mult}} = \texttt{function mult(x : nat, y : nat) : nat} \Leftarrow$
```
if(?0(x),
   0,
   if(even(x),
      mult(half(x), dbl(y)),
      plus(mult(half(x), dbl(y)), y))),
```

Abbildung 8.2: Prozedurdefinitionen für `plus`, `dbl`, `half`, `even` und `mult`.

$$
\begin{aligned}
\mathcal{U} \vdash\ & \texttt{if(}\underline{\texttt{let z:=plus(x, y) in dbl(z) end=0}}\texttt{, x, y)=0} && \rightarrow \\
& \underline{\texttt{if(let z:=plus(x, y) in dbl(z)=0 end, x, y)=0}} && \rightarrow \\
& \underline{\texttt{let z:=plus(x, y) in if(dbl(z)=0, x, y) end=0}} && \rightarrow \\
& \texttt{let z:=plus(x, y) in } \underline{\texttt{if(dbl(z)=0, x, y)=0}} \texttt{ end} && \rightarrow \\
& \texttt{let z:=plus(x, y) in if(dbl(z)=0, x=0, y=0) end}
\end{aligned}
$$

Im ersten Auswertungsschritt wird die rechte Seite der Gleichung

$$\texttt{let z:=plus(x, y) in dbl(z) end=0}$$

durch die Regel „*Distribute let argument*“ in den `let`-Rumpf gezogen. Anschließend wird der `let`-Term durch die Regel „*Distribute let body condition*“ aus der Bedingung des `if`-Terms geschoben. In den beiden letzten Schritten wird die rechte Seite der äußeren Gleichung durch die Regeln „*Distribute let argument*“ und „*Distribute argument*“ nach innen gezogen. Das Ergebnis dieser Auswertung ist der $\mathcal{L}$-normalisierte Term

$$\texttt{let z:=plus(x, y) in if(dbl(z)=0, x=0, y=0) end.}$$

□

8.3 Regeln für `case`-Terme

Zur symbolischen Auswertung von `case`-Termen definieren wir erstens Regeln, die das Verhalten eines Interpreters für `case`-Terme nachbilden, und zweitens Regeln, die zusätzliche symbolische Auswertungen ermöglichen. Konkret definieren wir zur Auswertung von `case`-Termen die folgenden Regeln:

⋮ ⋮
8. Keep branch
9. Skip negation
10. Skip alternatives
11. Merge branches
12. Skip condition
13. Evaluate condition
⋮ ⋮
16. Replace condition
17. Skip branch
18. Evaluate branch
⋮ ⋮

Die Definitionen dieser Regeln sind weitestgehend selbsterklärend. Lediglich die Definitionen der Regeln *Replace condition* und *Skip branch* erfordern eine etwas ausführlichere Erklärung. Wir beginnen jedoch unsere Definitionen der Regeln mit den Regeln „*Keep branch*“ und „*Evaluate condition*“ zur Nachbildung des Interpreters $eval_P$:

8. Keep branch

$$\frac{\mathtt{case}(b, cons_1 : r_1, \ldots, cons_i : r_i, \ldots, cons_l : r_l)}{r_i},$$

falls $b = cons(t^*)$ und ein $i \in \{1, \ldots, l\}$ mit $cons = cons_i$ existiert.

13. Evaluate condition

$$\frac{\mathtt{case}^{\,A}(b, r^*)}{\mathtt{case}^{\,A}(b', r^*)}, \qquad \text{falls } \mathcal{U} \vdash b \to b'.$$

Zusätzlich zu diesen Regeln definieren wir Regeln zur Elimination irrelevanter Fallunterscheidungen. Diese Regeln werden vor „*Evaluate condition*“ angewendet, da sich so klarere und zum Teil kürzere Auswertungen ergeben.

9. Skip negation

$$\frac{\mathtt{if}(\mathtt{if}(b, \mathtt{false}, \mathtt{true}), t_1, t_2)}{\mathtt{if}^{\,A_{\mathrm{std}}}(b, t_2, t_1)}$$

10. Skip alternatives

$$\frac{\mathtt{if}(b, \mathtt{true}, \mathtt{false})}{b}$$

11. Merge branches

$$\frac{\texttt{case}(b, cons_1 : r_1, \ldots, cons_i : r_i, \ldots, cons_l : r_l)}{\texttt{if}^{\,A_{\text{std}}}(\texttt{?}cons_i^{\,A_{\text{std}}}(b), r_i, r_{min(J)})},$$

falls $\mathsf{e}(b) \notin \mathcal{T}erm_{\texttt{bool}}(P)$ und $\mathsf{e}(r_j) \simeq_{\mathcal{U}} \mathsf{e}(r_{j'})$ für alle $j, j' \in J$ mit $J = \{1, \ldots, l\} \setminus \{i\}$.

12. Skip condition

$$\frac{\texttt{case}(b, cons_1 : r_1, \ldots, cons_l : r_l)}{r_1},$$

falls $\mathsf{e}(r_i) \simeq_{\mathcal{U}} \mathsf{e}(r_j)$ für alle $i, j \in \{1, \ldots, l\}$.

Da die Bedingung b eines case-Terms $\texttt{case}(b, t^*)$ nicht notwendigerweise zu einem Konstruktorgrundterm ausgewertet werden kann, ist eine Auswertung der Zweige des case-Terms erforderlich. Während dieser Auswertung dürfen wir die mit den Zweigen assoziierten Literale als gültig voraussetzen. Betrachten wir dazu ein Beispiel:

Beispiel 8.2. Sei der Term $t = \texttt{case}(\texttt{x}, \texttt{0} : \texttt{?0}(\texttt{x}), \texttt{succ} : \texttt{true})$ gegeben. Zur Auswertung des 0-Zweigs dürfen wir den Strukturtest $\texttt{?0}(\texttt{x})$ als gültig voraussetzen. Wir können daher den Term t zunächst zu $\texttt{case}(\texttt{x}, \texttt{0} : \texttt{true}, \texttt{succ} : \texttt{true})$ vereinfachen und anschließend mit Hilfe von „*Skip condition*“ zu true auswerten. □

Zur Auswertung von case-Zweigen definieren wir die Regel „*Evaluate branch*“. Aufgrund der in Abschnitt 8.2 definierten Regeln können wir für die Definition der Regel „*Evaluate branch*“ davon ausgehen, dass case-Terme $\mathcal{L}$-normalisiert sind und somit die Bedingungen von case-Termen case- und let-frei sind. Die definierte Anwendungsreihenfolge der Regeln gewährleistet weiterhin, dass ein Zweig t_i eines case-Terms $\texttt{case}(b, cons_1 : t_1, \ldots, cons_n : t_n)$ erst dann ausgewertet wird, wenn die Bedingung b des case-Terms vollständig ausgewertet ist.[2]

18. Evaluate branch

$$\frac{\texttt{case}^{\,A}(b, cons_1 : r_1, \ldots, cons_i : t_i, \ldots, cons_n : t_n)}{\texttt{case}^{\,A}(b, cons_1 : r_1, \ldots, cons_i : t'_i, \ldots, cons_n : t_n)} \qquad \frac{}{\begin{array}{r|l} & \text{Inst.} \\ \hline A: & I \end{array}}$$

falls $\mathcal{U}_i \vdash t_i \to t'_i$ und $t_j = t_j \downarrow_{\mathcal{U}_j}$ für alle $j \in \{1, \ldots, i-1\}$ gilt. Hierbei sind die A-Umgebungen $\mathcal{U}_k$ mit $k \in \{1, \ldots, n\}$ wie folgt definiert:

	Hyp_L	Hyp_I
$\mathcal{U}_k$:	$H_\text{L} \cup \{\texttt{?}(cons_k, \mathsf{e}(b))\}$	H_I^k

mit

$$H_\text{I}^k := \begin{cases} H_\text{I} & \text{falls } I = \bot, \\ H_\text{I} \cup \{\texttt{?}(cons_k, \mathsf{e}(b))\} & \text{falls } I = \top. \end{cases}$$

[2] Die in der Regel verwendeten Instanzhypothesenmengen H_I^k und das Instanz-Flag I ignorieren wir für den Augenblick. Wir kommen auf ihre Bedeutung in Kapitel 11 zu sprechen.

Die aktuell definierten Regeln sind noch nicht ausreichend, um `case`-Terme in befriedigender Weise symbolisch auszuwerten. Grund hierfür ist, dass die Informationen, die durch die lokalen und globalen Hypothesen gegeben sind, für die symbolische Auswertung von `case`-Termen nicht genügend genutzt werden. Wir definieren daher die beiden Regeln „*Replace condition*“ und „*Skip branch*“. Betrachten wir zunächst ein Beispiel:

Beispiel 8.3. Sei P das Programm $\langle D_{\texttt{tree}}, D_{\texttt{elem}} \rangle$, wobei $D_{\texttt{tree}}$ und $D_{\texttt{elem}}$ die Typoperator- und Prozedurdefinition aus Abbildung 3.3 bezeichnen. Betrachten wir nun den Term

```
if(?node(t),
   case(t,
        nil : elem(v,r),
        leaf : elem(v,r),
        node : elem(v,t)),
   false).
```
(8.1)

Mit Hilfe unserer aktuell definierten Regeln können wir den Term nicht weiter auswerten, obwohl er offensichtlich äquivalent zu

```
if(?node(t),elem(v,t),false)
```
(8.2)

ist. Dies hat zur Folge, dass nachfolgende Beweisschritte auf einem unnötig komplizierten Term arbeiten müssen und Beweisschritte für die irrelevanten `nil`- und `leaf`-Zweige durchgeführt werden. Da jedoch aufgrund der lokalen Hypothese `?node(t)` auch die Strukturgleichung `t=node(left(t),right(t))` gilt, können wir Term (8.1) durch den folgenden Term ersetzen:

```
if(?node(t),
   case(node(left(t),right(t)),
        nil : elem(v,r),
        leaf : elem(v,r),
        node : elem(v,t)),
   false).
```
(8.3)

Dieser Term kann dann durch Anwendung der Regel „*Keep branch*“ zu (8.2) ausgewertet werden. □

Wie das Beispiel zeigt, ist es sinnvoll, die Bedingung b eines `case`-Terms durch einen Term der Form $cons_i(sel_{i,1}(b), \ldots, sel_{i,n_i}(b))$ zu ersetzen, falls die Gültigkeit des Strukturtests $?cons_i(b)$ aus den lokalen und globalen Hypothesen folgt. Die Korrektheit der Ersetzung ergibt sich hierbei aus Lemma 3.27. Ist in der Menge $H_{\mathrm{L}} \cup H_{\mathrm{G}}$ eine Gleichung der Form $b{=}cons(t_1, \ldots, t_n)$ enthalten, so ist die Ersetzung der Bedingung b durch den Term $cons(t_1, \ldots, t_n)$ ebenfalls sinnvoll (und korrekt). Wir definieren daher folgende Regel:[3]

[3] Die Annotation A_{std} der Bedingung b' des `case`-Terms

$$\texttt{case}(\texttt{a}_{A_{\mathrm{std}}}(b'), cons_1 : r_1, \ldots, cons_l : r_l) \qquad (*)$$

in Regel „*Replace condition*“ ist irrelevant, da nach Anwendung der Regel „*Replace condition*“ sofort die Regel „*Keep branch*“ angewendet werden kann und der Term $(*)$ dadurch zu einem der Zweige vereinfacht wird. Die Annotation ist jedoch notwendig, da es sich bei b' um einen nicht-annotierten Term handelt. Jede andere Annotation wäre jedoch genauso geeignet.

16. Replace condition

$$\frac{\mathtt{case}^{\,A}(b, cons_1 : r_1, \ldots, cons_l : r_l)}{\mathtt{case}^{\,A}(\mathtt{a}_{A_{\mathrm{std}}}(b'), cons_1 : r_1, \ldots, cons_l : r_l)},$$

falls $\mathtt{e}(b) \notin \mathcal{T}erm_{\mathtt{bool}}(P)$ und

(1) $\mathtt{e}(b)\text{=}b' \in H_{\mathrm{L}} \cup H_{\mathrm{G}}$ und $b' = cons_i(t_1, \ldots, t_n)$, oder

(2) $\mathtt{?}cons_i(\mathtt{e}(b)) \in_{\simeq_{\mathcal{U}}} complete_P(H_{\mathrm{L}} \cup H_{\mathrm{G}})$ und $b' = cons_i(sel_{i,1}(\mathtt{e}(b)), \ldots, sel_{i,n_i}(\mathtt{e}(b)))$.

Betrachten wir ein weiteres Beispiel:

Beispiel 8.4. Sei P das Programm aus Beispiel 8.3. Betrachten wir nunmehr den Term

```
if(?node(t),
   false,
   case(t,
        nil : false,
        leaf : false,
        node : elem(v,t))).
```
(8.4)

Für diesen Term ist aufgrund der `if`-Bedingung `?node(t)` der `node`-Zweig des `case`-Terms irrelevant für die Auswertung des Terms. Wir dürfen daher den Term zu `false` vereinfachen. Eine solche Auswertung ist jedoch mit unseren aktuell definierten Regeln nicht möglich und es ergeben sich für Term (8.4) die gleichen Folgen wie für Term (8.1) aus Beispiel 8.3: Nachfolgende Beweisschritte müssen auf einem unnötig komplizierten Term arbeiten und es werden zusätzliche Beweisschritte für den irrelevanten `node`-Zweig durchgeführt. □

Das Beispiel zeigt, dass wir eine Regel benötigen, die die irrelevanten Zweige eines `case`-Terms eliminiert. Um die irrelevanten Zweige eines `case`-Terms

$$\mathtt{case}(b, cons_1 : r_1, \ldots, cons_l : r_l) \tag{8.5}$$

zu erkennen, müssen wir überprüfen, ob die mit den Zweigen assoziierten boolschen Terme $\mathtt{?}(cons_i, b)$ ungültig sind. Ist der Term $\mathtt{?}(cons_i, b)$ ungültig, so ist der entsprechende Zweig des `case`-Terms irrelevant. Enthält dann der `case`-Term (8.5) mehr als zwei `case`-Zweige, so können wir den Term (8.5) aufgrund der Ungültigkeit des Terms $\mathtt{?}(cons_i, b)$ zu

$$\mathtt{case}(b, \ldots, cons_{i-1} : r_{i-1}, cons_{i+1} : r_{i+1}, \ldots)$$

vereinfachen. Sind in (8.5) nur zwei `case`-Zweige enthalten, so können wir den `case`-Term durch einen der Zweige ersetzen. Durch die Elimination irrelevanter `case`-Zweige entstehen somit unvollständige `case`-Terme. Die Definition der Regel lautet wie folgt:

17. Skip branch

$$\frac{\texttt{case}^{\,A}(b, cons_1 : r_1, \ldots, cons_l : r_l)}{CASE(i, \texttt{case}^{\,A}(b, cons_1 : r_1, \ldots, cons_l : r_l))},$$

falls $\mathsf{e}(b) \notin \mathcal{T}erm_{\texttt{bool}}(P)$ und $\neg ?cons_i(\mathsf{e}(b)) \in_{\simeq_{\mathcal{U}}} complete_P(H_{\mathrm{L}} \cup H_{\mathrm{G}})$ gilt. Hierbei ist

$$CASE(i, \texttt{case}^{\,A}(b, cons_1 : r_1, \ldots, cons_l : r_l)) \quad =$$

$$\begin{cases} \texttt{case}^{\,A}(b, \ldots, cons_{i-1} : r_{i-1}, cons_{i+1} : r_{i+1}, \ldots) & \text{falls } l > 2, \\ r_1 & \text{falls } l = 2 \text{ und } i = 2, \\ r_2 & \text{falls } l = 2 \text{ und } i = 1. \end{cases}$$

8.4 Regeln für `let`-Terme

Wir definieren in diesem Abschnitt Regeln zur Auswertung von `let`-Termen. Hierbei gehen wir ähnlich vor wie für `case`-Termen, d.h. wir definieren erstens Regeln, die das Verhalten eines Interpreters nachbilden, und zweitens Regeln, die zusätzliche symbolische Auswertungen ermöglichen. Konkret definieren wir zur Auswertung von `let`-Termen die folgenden Regeln:

⋮ ⋮
19. Skip let
20. Replace let assignment
21. Evaluate let assignment
⋮ ⋮
24. Split let assignment
25. Evaluate let body
26. Distribute let body
⋮ ⋮

Die Definitionen der Regeln zur Auswertung von `let`-Termen sind wieder weitestgehend selbsterklärend . Lediglich die Definition der Regel „*Split let assignment*“ liegt nicht auf der Hand. Wir werden daher diese Regel ein wenig ausführlicher beschreiben.

19. Skip let

$$\frac{\texttt{let}\ x\texttt{:=}t\ \texttt{in}\ r\ \texttt{end}}{r}, \qquad \text{falls } x \notin fv(r).$$

Ist für einen `let`-Term `let` x`:=`t `in` r `end` die Variable x im Rumpf r enthalten, so werten wir zunächst den Term t aus, um möglichst einfache Variablenbindungen zu erhalten. Ist der Term t ausgewertet und enthält er keinen Prozeduraufruf mehr, ersetzen wir den `let`-Term durch den entsprechend instantiierten `let`-Rumpf. Enthält der Term t einen Prozeduraufruf, so ist ein Einsetzen der Variablenbindung

nicht sinnvoll, da sich dadurch Mehrfachauswertungen des Prozeduraufrufs ergeben können.

21. Evaluate let assignment

$$\frac{\texttt{let}^{\,A}\ x\texttt{:=}t\ \texttt{in}\ r\ \texttt{end}}{\texttt{let}^{\,A}\ x\texttt{:=}t'\ \texttt{in}\ r\ \texttt{end}}, \qquad \text{falls } \mathcal{U} \vdash t \to t'.$$

20. Replace let assignment

$$\frac{\texttt{let}\ x\texttt{:=}t\ \texttt{in}\ r\ \texttt{end}}{r[x/t]}$$

falls t keinen Prozeduraufruf einer Prozedur enthält.

Zur Auswertung des Rumpfs r eines `let`-Terms $\texttt{let}\ x\texttt{:=}t\ \texttt{in}\ r\ \texttt{end}$ müssen wir die aktuellen Variablenbindungen δ der A-Umgebung $\mathcal{U}$ um das Paar x/t erweitern. Wir definieren hierzu die folgende Regel.

25. Evaluate let body

$$\frac{\texttt{let}^{\,A}\ x\texttt{:=}t\ \texttt{in}\ r\ \texttt{end}}{\texttt{let}^{\,A}\ x\texttt{:=}t\ \texttt{in}\ r'\ \texttt{end}}, \qquad \frac{}{\mathcal{U}' : \ \delta[x/\mathrm{e}(t)]}\ \textit{Var.}$$

falls $\mathcal{U}' \vdash r \to r'$.[4]

Ist für einen Term $\texttt{let}\ x\texttt{:=}t\ \texttt{in}\ \texttt{case}(b, cons_1 : r_1, \ldots, cons_l : r_l)$ die Variable x lediglich in einem einzigen `case`-Zweig enthalten, so ist es sinnvoll, die Variablenbindung x=t in diesen `case`-Zweig hineinzuziehen und damit die Fallunterscheidung des `case`-Terms auch für die Variablenbindung x=t verfügbar zu machen. Wir definieren hierzu die folgende Regel.

26. Distribute let body

$$\frac{\texttt{let}^{\,A}\ x\texttt{:=}t\ \texttt{in}\ \texttt{case}^{\,B}(b, cons_1 : r_1, \ldots, cons_l : r_l)\ \texttt{end}}{\texttt{case}^{\,B}(b, cons_1 : r_1, \ldots, cons_i : (\texttt{let}^{\,B}\ x\texttt{:=}t\ \texttt{in}\ r_i\ \texttt{end}), \ldots, cons_l : r_l)},$$

falls $x \notin fv(b)$ und es existiert ein $i \in \{1, \ldots, l\}$ mit $x \in fv(r_i)$ und $x \notin fv(r_j)$ für alle $j \in \{1, \ldots, l\} \setminus \{i\}$.

Zur Definition der Regel „*Split let assignment*“ betrachten wir zunächst ein Beispiel.

Beispiel 8.5. Sei P das Programm $\langle D_{\texttt{plus}} \rangle$, wobei $D_{\texttt{plus}}$ die entsprechende Prozedurdefinition aus Abbildung 8.2 bezeichnet. Betrachten wir nun den Term

$$\begin{array}{l}\texttt{let x:=succ(plus(y,z))}\ \texttt{in}\\ \quad \texttt{case(x,0 : 0, succ : plus(x,z))}\ \texttt{end}.\end{array} \tag{8.6}$$

[4] Mit $\delta[x/t]$ bezeichnen wir die Termsubstitution $\{x/t\} \circ \delta$.

Diesen Term können wir mit den aktuell definierten Regeln nicht weiter vereinfachen, obwohl der Term offensichtlich auf den `succ`-Zweig des `case`-Terms reduziert werden kann. Die Regel „*Keep branch*“ ist jedoch auf den `case`-Term nicht anwendbar, da die Bedingung nicht mit einem Konstruktor beginnt. Eine mögliche Lösung dieses Problems wäre, den Term $\texttt{succ}(\texttt{plus}(\texttt{y},\texttt{z}))$ in den `case`-Term einzusetzen. Hierdurch würden sich aber unter Umständen unnötige Mehrfachauswertungen ergeben. Betrachtet man den Term (8.6) genauer, so erkennt man, dass es für die Anwendung der Regel „*Keep branch*“ nicht notwendig ist, den vollständigen Term in den `case`-Term einzusetzen. Vielmehr ist es ausreichend, lediglich den „Kontext“ $\texttt{succ}(\ldots)$ in den `case`-Term einzufügen:

$$\begin{array}{l}\texttt{let x}'\texttt{:=plus(y,z) in}\\ \quad\texttt{case succ(x}'\texttt{) of 0 : 0, succ : plus(succ(x}'\texttt{),z) end}\end{array}$$

Dieser Term ist äquivalent zu Term (8.6) und kann jetzt mit der Regel „*Keep branch*“ zu

$$\texttt{let x}'\texttt{:=plus(y,z) in plus(succ(x}'\texttt{),z) end}$$

vereinfacht werden. □

Allgemein ist es also sinnvoll, Konstruktoren aus der Variablenbindung x`:=`t eines `let`-Terms $\texttt{let}\ x\texttt{:=}t\ \texttt{in}\ r\ \texttt{end}$ herauszuziehen und in den Rumpf r einzusetzen. Dieses Einsetzen der Konstruktoren ermöglicht, wie in Beispiel 8.5 dargestellt, Regelanwendungen ohne Mehrfachauswertungen zu erzeugen. Um die entsprechende Regel für dieses Einsetzen der Konstruktoren zu definieren, führen wir *Konstruktorkontexte* ein:

Definition 8.6 (Konstruktorkontext)**.** Sei P ein Programm und $V \subseteq \mathcal{V}(\Omega(P))$ eine nicht-leere Menge von Termvariablen. Ein `let`- und `case`-freier A-Term $c \in \mathcal{ATerm}(P)$ heißt *Konstruktorkontext bzgl.* V, falls

(1) alle Funktionssymbole von c Konstruktoren sind, und

(2) alle Termvariablen $x \in V$ in c genau einmal vorkommen, und

(3) c mit einem Konstruktor beginnt.

□

Die Regel „*Split let assignment*“ zieht dann aus einer Variablenbindung eines `let`-Terms einen maximalen Konstruktorkontext heraus und setzt diesen in den Rumpf des `let`-Terms ein. Durch die Verwendung eines maximalen Konstruktorkontexts verhindern wir die wiederholte Anwendung der Regel für Terme wie beispielsweise

$$\texttt{let x:=succ(succ(}t\texttt{)) in }r\texttt{ end.}$$

Die Definition der Regel lautet wie folgt:

24. Split let assignment

$$\frac{\texttt{let}\ x\texttt{:=}t\ \texttt{in}\ r\ \texttt{end}}{\texttt{let}^{\ A_{\text{std}}}\ x_1\texttt{:=}t_1\ \texttt{in}\ \ldots\texttt{let}^{\ A_{\text{std}}}\ x_n\texttt{:=}t_n\ \texttt{in}\ r[x/c[x_1,\ldots,x_n]]\ \texttt{end}\ldots\texttt{end}},$$

falls eine Termvariablenmenge $V = \{x_1, \ldots, x_n\}$ und ein Konstruktorkontext c bzgl. V sowie Terme $t_1 \in \mathcal{Term}_{\tau_1}(P), \ldots, t_n \in \mathcal{Term}_{\tau_n}(P)$ existieren, so dass die folgenden Bedingungen erfüllt sind:

(1) $t = c[x_1/t_1, \ldots, x_n/t_n]$,

(2) $x_i = \mathit{Var}_{\tau_i}(\mathit{fv}(r) \cup \mathit{bv}(r) \cup \{x_1, \ldots, x_{i-1}\})$ für alle $i \in \{1, \ldots, n\}$,

(3) $t_i = f_i(s_i^*)$ mit $f_i \in \Sigma(P) \setminus \Sigma^{\text{cons}}(P)$ für alle $i \in \{1, \ldots, n\}$.

Die Anwendungsbedingungen (1) und (2) der Regel „*Split let assignment*“ gewährleisten die Korrektheit der Regel und die Anwendungsbedingung (3) stellt sicher, dass es sich bei c um einen maximalen Konstruktorkontext handelt. Die Regel wird vor der Auswertung des Rumpfs eines `let`-Terms durch die Regel „*Evaluate let body*“ angewendet. Damit gewährleisten wir, dass die Variablenbindung δ der A-Umgebung $\mathcal{U}$ ausschließlich Paare der Form

$$x/f(\ldots) \text{ mit } f \notin \Sigma^{\text{cons}}(P). \tag{8.7}$$

enthält. In einigen der nachfolgenden Regeldefinitionen überprüfen wir, ob ein Teilterm t' des aktuell auszuwertenden Terms mit einem Konstruktor beginnt. Aufgrund der Eigenschaft (8.7) müssen wir dann bei dieser Überprüfung die aktuelle Variablenbindung δ nicht berücksichtigen.

8.5 Regeln für Gleichungen

Wir definieren in diesem Abschnitt Regeln zur Auswertung von Gleichungen. Hierzu definieren wir zunächst eine Regel zur Implementierung der Reflexivität. Anschließend führen wir die Regeln „*Constructor uniqueness*“ und „*Constructor injectivity*“ ein, um das Auswertungsverhalten von Gleichungen bzgl. Konstruktoren zu definieren. Abschließend betrachten wir die „*Replace*“-Regeln, die zusätzliche Anwendungen der Regeln „*Constructor uniqueness*“ und „*Constructor injectivity*“ ermöglichen. Die Anwendungsreihenfolge der Regeln ist wie folgt festgelegt:

⋮	⋮
27.	*Reflexivity*
28.	*Constructor uniqueness*
29.	*Constructor injectivity*
⋮	⋮
39.	*Replace left equality argument*
40.	*Replace right equality argument*
⋮	⋮

Wir beginnen die Definitionen der Regeln mit der „*Reflexivity*“-Regel:

27. Reflexivity

$$\frac{l{=}r}{\texttt{true}}, \qquad \text{falls } \mathsf{e}(l{=}r) \in \mathcal{A}t(P) \text{ und } \mathsf{e}(l) \simeq_{\mathcal{U}} \mathsf{e}(r).$$

Betrachten wir nun Gleichungen, deren linke und rechte Seiten nicht $\simeq_{\mathcal{U}}$-äquivalent sind. Beginnen die linke und die rechte Seite einer Gleichung $l{=}r$ mit verschiedenen Konstruktoren, so bezeichnen l und r verschiedene Werte und wir können die Gleichung direkt durch `false` ersetzen. Für Gleichungen wie

$$\texttt{x=succ(x)}$$

ist eine solche Auswertung zu `false` ebenfalls möglich. Der Grund hierfür ist, dass die linke Seite ein *echter Konstruktorteilterm* der rechten Seite ist. Hierbei bezeichnen wir einen Term s als echten Konstruktorteilterm eines Terms t falls $t >^{+}_{\Sigma^{\text{cons}}(P)} s$ gilt, wobei $>^{+}_{\Sigma^{\text{cons}}(P)}$ die transitive Hülle der Relation $>_{\Sigma^{\text{cons}}(P)} \subseteq \mathcal{T}erm(P) \times \mathcal{T}erm(P)$ mit

$$\begin{gathered} t >_{\Sigma^{\text{cons}}(P)} s \\ \text{gdw.} \\ t = cons(\ldots, s, \ldots) \text{ mit } cons \in \Sigma^{\text{cons}}(P) \end{gathered} \tag{8.8}$$

bezeichnet. Es gilt dann für alle Terme t und s die folgende, leicht nachzuweisende Implikation:

$$t >^{+}_{\Sigma^{\text{cons}}(P)} s \quad \Longrightarrow \quad P \models \texttt{all } \nu^*, x^*\ t{\neq}s.$$

Damit definieren wir jetzt die Regel zur Auswertung von Gleichungen mit unterschiedlichen linken und rechten Seiten:

28. Constructor uniqueness

$$\frac{l{=}r}{\texttt{false}},$$

falls

(1) $\mathsf{e}(l) = cons(t^*)$, $\mathsf{e}(r) = cons'(s^*)$ sowie $cons \neq cons'$ und $cons, cons' \in \Sigma^{\text{cons}}(P)$ oder

(2) $\mathsf{e}(l) >^{+}_{\Sigma^{\text{cons}}(P)} \mathsf{e}(r)$ oder $\mathsf{e}(r) >^{+}_{\Sigma^{\text{cons}}(P)} \mathsf{e}(l)$.

Mit den aktuell definierten Regeln zur Auswertung von Gleichungen ist es noch nicht möglich Gleichungen wie

$$\texttt{add(0, empty)=add(0, n)} \tag{8.9}$$

oder

$$\texttt{add(0, empty)=add(succ(0), empty)} \tag{8.10}$$

zu vereinfachen. Aufgrund der Injektivität von Konstruktoren (Lemma 3.26) wissen wir jedoch, dass eine Gleichung der Form

$$cons(t_1, \ldots, t_n){=}cons(r_1, \ldots, r_n)$$

$$\begin{array}{lll} \mathcal{U} \vdash & \underline{\texttt{if(0=0)}}\texttt{,empty=n,false)} & \rightarrow \\ & \underline{\texttt{if(true,empty=n,false)}} & \rightarrow \\ & \texttt{empty=n.} & \end{array}$$

Abbildung 8.3: Auswertung der Konjunktion (8.11)

$$\begin{array}{lll} \mathcal{U} \vdash & \texttt{if(}\underline{\texttt{0=succ(0)}}\texttt{,empty=empty,false)} & \rightarrow \\ & \underline{\texttt{if(false,empty=empty,false)}} & \rightarrow \\ & \texttt{false} & \end{array}$$

Abbildung 8.4: Auswertung der Konjunktion (8.12)

genau dann gültig ist, wenn die Argumentgleichungen

$$t_1 \texttt{=} r_1, \quad \ldots, \quad t_n \texttt{=} r_n$$

gültig sind. Wir können daher die Gleichungen (8.9) und (8.10) durch die Konjunktionen

$$\texttt{if(0=0,empty=n,false)} \tag{8.11}$$

bzw.

$$\texttt{if(0=succ(0),empty=empty,false)} \tag{8.12}$$

ersetzen. Die Auswertungen dieser Konjunktionen sind in den Abbildungen 8.3 und 8.4 dargestellt. Mit Hilfe der Zerlegung in die Argumentgleichungen kann also Gleichung (8.9) auf die eigentlich relevante Gleichung `empty=n` reduziert werden und Gleichung (8.10) vollständig ausgewertet werden. Eine Zerlegung in die Argumentgleichungen ist somit in beiden Fällen gewinnbringend. Wir definieren allgemein die folgende Regel.

29. Constructor injectivity

$$\frac{cons(t_1,\ldots,t_n) =^{A} cons(r_1,\ldots,r_n)}{\mathbf{a}_{A_{\text{std}}}(AND(x_1,\ldots,x_n))[x_1/t_1 =^{A} r_1,\ldots,x_n/t_n =^{A} r_n]^{5}},$$

Aufgrund der bisher eingeführten Regeln zur Auswertung von Gleichungen ist es günstig, wenn sowohl die linke als auch die rechte Seite einer Gleichung mit einem Konstruktor beginnt. Wir definieren daher zwei Regeln, die die linke bzw. rechte Seite einer Gleichung durch einen Term der Form $cons_i(sel_{i,1}(t),\ldots,sel_{i,n_i}(t))$ ersetzt, sofern die entsprechende Seite nicht bereits mit einem Konstruktor beginnt:

[5] Durch die Verwendung des Terms

$$\mathbf{a}_{A_{\text{std}}}(AND(x_1,\ldots,x_n))[x_1/t_1 =^{A} r_1,\ldots,x_n/t_n =^{A} r_n]$$

wird sichergestellt, dass alle Gleichungen $t_i \texttt{=} r_i$ ihre Annotationen aus der ursprünglichen Gleichung

$$cons(t_1,\ldots,t_n) =^{A} cons(r_1,\ldots,r_n)$$

übernehmen. Dies wäre durch die Verwendung des Terms $\mathbf{a}_{\mathtt{A}_{\text{std}}}(AND(\mathtt{t_1=r_1},\ldots,\mathtt{t_1=r_n}))$ nicht gewährleistet.

$$\left.\begin{array}{l} \dfrac{l{=}r}{cons_i(sel_{i,1}(l),\ldots,sel_{i_{n_i}}(l)){=}r}, \\ \text{falls der Term } l \text{ nicht mit einem Konstruktor beginnt und die folgenden Bedingungen erfüllt sind:} \\ \quad (1)\ \texttt{?}cons_i(\mathbf{e}(l)) \in complete_P(H_\mathrm{L} \cup H_\mathrm{G}) \text{ und} \\ \quad (2)\ r = cons_j(\ldots) \text{ oder } \texttt{?}cons_i(\mathbf{e}(r)) \in complete_P(H_\mathrm{L} \cup H_\mathrm{G}). \end{array}\right\} \quad (8.13)$$

$$\left.\begin{array}{l} \dfrac{l{=}r}{l{=}cons_i(sel_{i,1}(r),\ldots,sel_{i_{n_i}}(r))}, \\ \text{falls der Term } r \text{ nicht mit einem Konstruktor beginnt und die folgenden Bedingungen erfüllt sind:} \\ \quad (1)\ \texttt{?}cons_i(\mathbf{e}(r)) \in complete_P(H_\mathrm{L} \cup H_\mathrm{G}) \text{ und} \\ \quad (2)\ l = cons_j(\ldots). \end{array}\right\} \quad (8.14)$$

Die Anwendungsbedingungen (1) der Regeln (8.13) und (8.14) stellen sicher, dass die Gültigkeit eines entsprechenden Strukturtests und damit die Gültigkeit einer entsprechenden Strukturgleichung aus der globalen und der lokalen Hypothesenmenge folgt. Diese Anwendungsbedingungen gewährleisten somit die Korrektheit der Regeln. Die Anwendungsbedingungen (2) schränken die Regeln auf jeweils die Fälle ein, für die nach einer möglichen Anwendung der beiden Regeln sowohl die linke als auch die rechte Seite der Gleichung mit einem Konstruktor beginnen. Betrachten wir dazu ein Beispiel:

Beispiel 8.7. Sei der Term $\texttt{if(?succ(x),if(?succ(y),x=y,true),true)}$ gegeben. Dieser Term kann mit Hilfe der Regeln (8.13) und (8.14) wie folgt ausgewertet werden:

$$\begin{array}{lll} \mathcal{U} \vdash \texttt{if(?succ(x),if(?succ(y),}\underline{\texttt{x=y}}\texttt{,true),true)} & \rightarrow \\ \quad \texttt{if(?succ(x),if(?succ(y),}\underline{\texttt{succ(pred(x))=y}}\texttt{,true),true)} & \rightarrow \\ \quad \texttt{if(?succ(x),if(?succ(y),}\underline{\texttt{succ(pred(x))=succ(pred(y))}}\texttt{,true),true)} & \rightarrow \\ \quad \texttt{if(?succ(x),if(?succ(y),}\underline{\texttt{pred(x)=pred(y)}}\texttt{,true),true).} \end{array}$$

Hierbei ist zu beachten, dass die Regel (8.13) im ersten Auswertungsschritt anwendbar ist, obwohl die rechte Seite nicht mit einem Konstruktor beginnt. Dies wird durch den zweiten Teil der Anwendungsbedingung (2) gewährleistet. Dieser Teil der Anwendungsbedingung stellt sicher, dass die Regel (8.13) auch dann angewendet wird, falls die rechte Seite erst nach einer anschließenden Anwendung der Regel (8.14) mit einem Konstruktor beginnt. Dabei wird die feste Reihenfolge der Regelanwendungen ausgenutzt. □

Für den Fall, dass für eine Gleichung $l{=}r$ mit $l, r \in \mathcal{T}erm_{\zeta[\ldots]}$ der Typoperator ζ lediglich einen einzigen Konstruktor definiert, sind die Regeln (8.13) und (8.14) immer anwendbar. Betrachten wir dazu ein Beispiel.

Beispiel 8.8. Sei P das Programm $\langle D_{\texttt{state}} \rangle$, wobei $D_{\texttt{state}}$ die folgende Typoperatordefinition bezeichnet:

$$\begin{array}{l} \texttt{structure state} \Leftarrow \\ \quad \texttt{st(rega : nat, regb : nat).} \end{array}$$

Da $\texttt{?st(x)}, \texttt{?st(y)} \in \mathit{complete}_P(\emptyset)$ gilt, kann der Term `x=y` aufgrund der Regeln (8.13) und (8.14) wie folgt ausgewertet werden:

$$\begin{array}{lll} \mathcal{U} \vdash & \underline{\texttt{x=y}} & \rightarrow \\ & \underline{\texttt{state(rega(x),regb(x))=y}} & \rightarrow \\ & \underline{\texttt{state(rega(x),regb(x))=state(rega(y),regb(y))}} & \rightarrow \\ & \underline{\texttt{if(rega(x)=rega(y),regb(x)=regb(y),false)}} & \end{array}$$

□

Auswertungen wie in Beispiel 8.8 haben sich in unseren Fallstudien als ungünstig erwiesen, da sie zur Folge haben, dass Gleichungen unnötigerweise in ihre Komponentengleichungen zerlegt werden. Wir beschränken daher die Anwendung der Regeln (8.13) und (8.14) für den Fall, dass der Typoperator ζ nur einen einzigen Konstruktor definiert, auf die Fälle, in denen eine der Komponentengleichungen l_i=r_i der Gleichung $\mathit{cons}(l_1,\ldots,l_n)$= $\mathit{cons}(r_1,\ldots,r_n)$ zu `true` oder `false` ausgewertet werden kann. Damit ist sichergestellt, dass durch die Ersetzung der linken und rechten Seiten einer Gleichung l=r durch die Terme $\mathit{cons}(l_1,\ldots,l_n)$ und $\mathit{cons}(r_1,\ldots,r_n)$ und einer anschließenden Zerlegung in Komponentengleichungen die ursprüngliche Gleichung l=r wirklich vereinfacht wird.[6] Dies führt zu folgender Regeldefinition:

39. Replace left equality argument

$$\frac{l =^{A} r}{\mathit{cons}_i^{\,A_{\mathrm{std}}}(\mathit{sel}_{i,1}^{\,A_{\mathrm{std}}}(l),\ldots,\mathit{sel}_{i_{n_i}}^{\,A_{\mathrm{std}}}(l)) =^{A} r},$$

falls

(1) $l \neq \mathit{cons}_k(\ldots)$,

(2) $\texttt{?}\mathit{cons}_i(\mathsf{e}(l)) \in_{\simeq_{\mathcal{U}}} \mathit{complete}_P(H_{\mathrm{L}} \cup H_{\mathrm{G}})$,

(3) $r = \mathit{cons}_j(\ldots)$ oder $\texttt{?}\mathit{cons}_i(\mathsf{e}(r)) \in \mathit{complete}_P(H_{\mathrm{L}} \cup H_{\mathrm{G}})$ und

(4) falls $l, r \in \mathcal{ATerm}_{\zeta[\ldots]}(P)$ und $|\Sigma_\zeta^{\mathrm{cons}}(P)| = 1$ gilt, so ist

$$\mathsf{a}_{A_{\mathrm{std}}}(\mathit{sel}_{i,j}(\mathsf{e}(l))\texttt{=}\mathit{sel}_{i,j}(\mathsf{e}(r)))\downarrow_{\mathcal{U}'} \in \{\texttt{false},\texttt{true}\}.^{7}$$

Hierbei bezeichnet $\mathcal{U}'$ die folgende A-Umgebung:

	Proz.
$\mathcal{U}'$:	$\emptyset$

[6] Aus Effizienzgründen sollte die symbolische Auswertung der Komponentengleichungen keine Prozeduren auswerten und auch keine Quantifikationen sowie keine „*Functionality*"-Regeln verwenden (siehe die Annotation A_{std} und die Definition der A-Umgebung $\mathcal{U}'$ in den Regeln 39 und 40).

[7] $\Sigma_\zeta^{\mathrm{cons}}(P)$ bezeichnet die Menge der Konstruktoren von ζ in P.

40. Replace right equality argument

$$\frac{l =^{A} r}{l =^{A} cons_i\,{}^{A_{\mathrm{std}}}(sel_{i,1}\,{}^{A_{\mathrm{std}}}(r), \ldots, sel_{i_{n_i}}\,{}^{A_{\mathrm{std}}}(r))},$$

falls

(1) $l \neq cons_k(\ldots)$,

(2) $?cons_i(\mathbf{e}(r)) \in_{\simeq_{\mathcal{U}}} complete_P(H_{\mathrm{L}} \cup H_{\mathrm{G}})$,

(3) $l = cons_j(\ldots)$ und

(4) falls $l, r \in \mathcal{ATerm}_{\zeta[\ldots]}(P)$ und $|\Sigma^{\mathrm{cons}}(\zeta)| = 1$ gilt, so ist

$$\mathbf{a}_{A_{\mathrm{std}}}(sel_{i,j}(\mathbf{e}(l)) {=} sel_{i,j}(\mathbf{e}(r)))\downarrow_{\mathcal{U}'} \in \{\mathtt{false}, \mathtt{true}\}.$$

Hierbei bezeichnet $\mathcal{U}'$ die folgende A-Umgebung:

	Proz.
$\mathcal{U}'$:	$\emptyset$

Wir wenden die Regeln „*Replace left equality argument*“ und „*Replace right equality argument*“ auf eine Gleichung $l{=}r$ erst dann an, wenn die Terme l und r ausgewertet sind. Hintergrund ist, dass die Regeln „*Replace left equality argument*“ und „*Replace right equality argument*“ nicht nach jedem einzelnen Auswertungsschritt in einem der Terme l und r erneut überprüft werden sollen. Abschließend wollen wir die Nützlichkeit der „*Replace*“-Regeln anhand eines weiteren Beispiels demonstrieren.

Beispiel 8.9. Sei P das Programm $\langle D_{\mathtt{plus}} \rangle$, wobei $D_{\mathtt{plus}}$ die Prozedurdefinition aus Abbildung 8.2 bezeichnet. Wir wollen das folgende Lemma mittels Induktion beweisen:

$$\begin{array}{l}\mathtt{lemma\ plus\_neutral} \Leftarrow \mathtt{all\ x : nat} \\ \quad \mathtt{plus(x, 0){=}x.}\end{array}$$

Hierbei verwenden wir die folgende Sequenz als Induktionsschritt:

$$\{\mathtt{?succ(x)}\}, \{\mathtt{plus(pred(x), 0){=}pred(x)}\} \vdash \mathtt{plus(x, 0){=}x.}$$

Die Gültigkeit dieses Induktionsschritts beweisen wir dadurch, dass wir den Goal-Term durch symbolische Auswertung auf die Induktionshypothese reduzieren. Für den Goal-Term erhalten wir die folgende symbolische Auswertung:[8]

$$\begin{array}{lll} \mathcal{U} \vdash & \underline{\mathtt{plus(x, 0)}}\mathtt{{=}x} & \rightarrow \\ & \mathtt{if(}\underline{\mathtt{?0(x)}}\mathtt{, 0, succ(plus(pred(x), 0)))=x} & \rightarrow \\ & \underline{\mathtt{if(false, 0, succ(plus(pred(x), 0)))=x}} & \rightarrow \\ & \mathtt{succ(plus(pred(x), 0))=x.} & \end{array}$$

Durch Anwendung der Regel „*Replace right equality argument*“ können wir jetzt den Term auf die Induktionshypothese reduzieren und somit die Gültigkeit des Induktionsschritts beweisen:

[8] Hierbei gehen wir davon aus, dass wir Prozeduraufrufe durch die instantiierten Prozedurrümpfe ersetzen und dass die globale Hypothesenmenge der A-Umgebung $\mathcal{U}$ durch die Hypothesenmenge des Induktionsschritts gegeben ist.

$$\begin{array}{lll} \mathcal{U} \vdash & \texttt{succ(plus(pred(x),0))=x} & \rightarrow \\ & \underline{\texttt{succ(plus(pred(x),0))=succ(pred(x))}} & \rightarrow \\ & \texttt{plus(pred(x),0)=pred(x)}. & \end{array}$$

□

Anmerkung 8.10. Vergleicht man die beiden Regeln „*Replace left equality argument*“ und „*Replace right equality argument*“ mit der Regel „*Replace condition*“ aus Abschnitt 8.3 so stellt man fest, dass wir für die Regeln „*Replace left equality argument*“ und „*Replace right equality argument*“ ausschließlich Ersetzungen aufgrund von Strukturtests durchführen. Existiert eine globale oder lokale Hypothese der Form l=*cons*(...), so wird diese Hypothese von den Regeln nicht berücksichtigt. Betrachten wir beispielsweise das Programm $P = \langle D_{\texttt{plus}} \rangle$. Eine Auswertung des Terms

$$\texttt{if(x=succ(plus(x,z)),x=succ(plus(y,z)),true)}$$

ist mit Hilfe unserer aktuell definierten Regeln nicht möglich. Prinzipiell spricht nichts dagegen, die linke Seite der Gleichung `x=succ(plus(y,z))` durch den Term `succ(plus(x,z))` zu ersetzen und den Term insgesamt zu

$$\texttt{if(x=succ(plus(x,z)),plus(x,z)=plus(y,z),true)}$$

zu vereinfachen. Da während der Auswertung von `x=succ(plus(y,z))` der Term `x=succ(plus(x,z))` lediglich als nicht-annotierte lokale Hypothese zur Verfügung steht, stellt sich jedoch die Frage, wie der Term `succ(plus(x,z))` für die Ersetzung annotiert werden sollte. Diese Annotationen sind im Gegensatz zu den Annotationen der Bedingung des `case`-Terms in „*Replace condition*“ relevant, da der eingesetzte Term nicht notwendigerweise durch spätere Auswertungen verschwindet. Da in keiner unserer Fallstudien Ersetzungen aufgrund von Gleichungen der Form l=*cons*(...) notwendig waren, verzichten wir darauf, dieses Annotations-Problem genauer zu analysieren. □

8.6 Regeln für Strukturprädikate und Selektoren

Wir definieren in diesem Abschnitt die Regeln zur Auswertung von Strukturprädikaten und Selektoren. Hierzu betrachten wir zunächst Regeln, die die Arbeitsweise eines Interpreters nachbilden. Anschließend definieren wir die Regeln „*Structure equation to structure test*“ und „*Replace structure predicate argument*“. Die erste Regel ersetzt Strukturgleichungen durch die äquivalenten Strukturtests. Die zweite Regel ersetzt das Argument eines Strukturtests auf Basis der lokalen und globalen Hypothesen, so dass der Strukturtest anschließend zu `true` bzw. `false` ausgewertet werden kann. Die Anwendungsreihenfolge der Regeln ist wie folgt gegeben:

⋮ ⋮

30. Affirmative structure test
31. Negative structure test
32. Appropriate selector
33. Structure equation to structure test
34. Replace structure predicate argument

⋮ ⋮

Wir beginnen die Definitionen der Regeln mit den Regeln zur Nachbildung eines Interpreters:

30. Affirmative structure test

$$\frac{?cons(cons(r^*))}{\texttt{true}}$$

31. Negative structure test

$$\frac{?cons(cons'(r^*))}{\texttt{false}}, \qquad \text{falls } cons \neq cons'.$$

32. Appropriate selector

$$\frac{sel_{i,j}(cons_i(t_1, \ldots, t_j, \ldots, t_n))}{t_j}$$

Strukturgleichungen $t{=}cons_i(sel_{i,1}(t), \ldots, sel_{i,m_i}(t))$ sind zur symbolischen Auswertung äußerst ungeeignet, da der Term t mehrfach in der Gleichung vorkommt und so Mehrfachauswertungen des Terms t erforderlich sind. Wir ersetzen daher Strukturgleichungen durch die äquivalenten Strukturtests.

33. Structure equation to structure test

$$\frac{t{=}cons_i(sel_{i,1}(t_1), \ldots, sel_{i,m_i}(t_n))}{?cons^{\,A_{std}}(t_1)},$$

falls für alle $i \in \{1, \ldots, n\}$ gilt $t_1 \simeq_{\mathcal{U}} t_n$.

Ist in der lokalen oder globalen Hypothesenmenge eine Gleichung der Form $t = cons'(t_1, \ldots, t_n)$ enthalten, so ist es sinnvoll das Argument t eines Strukturtests $?cons(t)$ durch $cons'(t_1, \ldots, t_n)$ zu ersetzen. Durch diese Ersetzung ermöglichen wir die Anwendung der Regel „*Affirmative structure test*“ bzw. „*Negative structure test*“ und so die Auswertung des Strukturtests zu `true` bzw. `false`. Für Ersetzungen dieser Art definieren wir daher die folgende Regel:

34. Replace structure predicate argument

$$\frac{?cons(l)}{?cons(\mathbf{a}_{A_{\mathrm{std}}}(r))},$$

falls $l{=}r \in_{\simeq_{\mathcal{U}}} complete_P(H_{\mathrm{L}} \cup H_{\mathrm{G}})$ und $r = cons'(t^*)$.

8.7 Regel für Funktionsargumente

Wir haben bereits für einige unserer Beispiele vorausgesetzt, dass die Argumente von Funktionsaufrufen ausgewertet werden. Wir definieren hierfür nun eine entsprechende Regel.

36. Evaluate argument

$$\frac{f\,{}^{A}(t_1,\ldots,t_i,\ldots,t_n)}{f\,{}^{A}(t_1,\ldots,t'_i,\ldots,t_n)},$$

falls $f \notin \Sigma^{\text{case}}(P)$, $\mathcal{U}' \vdash t_i \to t'_i$ und $t_j = (t_j{\downarrow_{\mathcal{U}'}})$ für alle $j \in \{1,\ldots i-1\}$ gilt. Hierbei ist die A-Umgebung $\mathcal{U}'$ wie folgt definiert:

	Ktx.
$\mathcal{U}'$:	$C \cup C'$

mit $C' = \{f\}$ falls $f \in \Sigma^{\text{proc}}(P)$ und $C' = \emptyset$ sonst.

Diese Regel wird vor den Regeln aus Abschnitt 8.2 zur $\mathcal{L}$-Normalisierung angewendet.

8.8 Regeln zur Anordnung von Bedingungen

Wir definieren in diesem Abschnitt Regeln um die Atome in boolschen `if`-Termen anzuordnen. Wir definieren hierzu die folgenden Regeln:

⋮ ⋮
79. Move then-part atom
80. Move else-part atom
81. Commute then-part atom
82. Commute else-part atom
⋮ ⋮

Für Atome a und b repräsentieren wir die Konjunktion $a \wedge b$ durch den `if`-Term

$$\texttt{if}(a, b, \texttt{false}). \tag{8.15}$$

Aufgrund der Kommutativität der Konjunktion können wir die Atome a und b innerhalb des Term (8.15) vertauschen. Das bedeutet, dass wir die Konjunktion $a \wedge b$ auch durch den Term

$$\texttt{if}(b, a, \texttt{false}) \tag{8.16}$$

ausdrücken können. Für die symbolische Auswertung kann es von entscheidender Bedeutung sein, durch welchen der beiden Terme die Konjunktion repräsentiert wird. Betrachten wir dazu ein Beispiel:[9]

Beispiel 8.11. Sei P das Programm $\langle D_{\texttt{even}}\rangle$, wobei $D_{\texttt{even}}$ die entsprechende Prozedurdefinition aus Abbildung 8.2 bezeichnet. Die Konjunktion $\texttt{even(x)} \wedge \texttt{?0(x)}$ können wir sowohl durch den Term

$$\texttt{if}(\texttt{even(x)}, \texttt{?0(x)}, \texttt{false}) \tag{8.17}$$

[9] Für diesen Abschnitt gehen wir davon aus, dass ein Prozeduraufrufe $f(t^*)$ ausgewertet werden darf, falls er auf eine Basisfall der Prozedur reduziert werden kann. Diese Heuristik wird durch die in Abschnitt 9.3 definierte Regel „*Execute procedure call (non recusive cases)*“ realisiert.

als auch durch den Term

$$\texttt{if}(\texttt{?0}(\texttt{x}), \texttt{even}(\texttt{x}), \texttt{false}) \tag{8.18}$$

repräsentieren. Term (8.17) können wir durch symbolische Auswertung nicht vereinfachen. Für Term (8.18) ergibt sich hingegen die folgende symbolische Auswertung:

$$\begin{array}{lll} \mathcal{U} \vdash & \texttt{if}(\texttt{?0}(\texttt{x}), \underline{\texttt{even}(\texttt{x})}, \texttt{false}) & \rightarrow \\ & \texttt{if}(\texttt{?0}(\texttt{x}), \texttt{if}(\underline{\texttt{?0}(\texttt{x})}, \texttt{true}, \ldots), \texttt{false}) & \rightarrow \\ & \texttt{if}(\texttt{?0}(\texttt{x}), \underline{\texttt{if}(\texttt{true}, \texttt{true}, \ldots)}, \texttt{false}) & \rightarrow \\ & \underline{\texttt{if}(\texttt{?0}(\texttt{x}), \texttt{true}, \texttt{false})} & \rightarrow \\ & \texttt{?0}(\texttt{x}). & \end{array}$$

Die Konjunktion $\texttt{even}(\texttt{x}) \wedge \texttt{?0}(\texttt{x})$ kann also, falls sie durch den „richtigen“ `if`-Term repräsentiert wird, durch symbolische Auswertung auf das Atom $\texttt{?0}(\texttt{x})$ reduziert werden. □

Es stellt sich also das Problem, wie wir für eine Konjunktion beurteilen können, welche der Repräsentationen für die symbolische Auswertung besser geeignet ist. Da für Term (8.15) das Atom a als lokale Hypothese zur Auswertung von Atom b verwendet wird und für Term (8.16) das Atom b als lokale Hypothese zur Auswertung des Atoms a, lässt sich das Problem auf die Fragestellung reduzieren, welches der beiden Atom a und b eine bessere lokale Hypothese zur Auswertung des anderen Atoms darstellt. In Beispiel 8.11 war es günstiger den Strukturtest $\texttt{?0}(\texttt{x})$ als lokale Hypothese zur Auswertung von $\texttt{even}(\texttt{x})$ zu verwenden, da aufgrund des Strukturtests die Prozedur ausgewertet werden konnte. Allgemein sind Strukturtests ausgenommen mächtige lokale Hypothesen. Dies hat zwei Gründe:

- Strukturtests bzw. Strukturprädikate haben besondere Eigenschaften. Wir haben diese Eigenschaften in Lemma 3.27 zusammengefasst. Einige der von uns definierten Auswertungsregeln nutzen diese Eigenschaften explizit aus. Die Regeln „*Replace left equality argument*“ und „*Replace right equality argument*“ schließen beispielsweise aus der Gültigkeit des Strukturtests $\texttt{?}cons(t)$, dass t ein $cons$-Term ist. Strukturtests sind daher in besonderer Weise für die Anwendbarkeit von Regeln verantwortlich.
- Die Fallunterscheidungen in Prozeduren sind häufig auf Basis von Strukturtests definiert. Die Auswertung von Prozeduren ist daher stark von den Strukturtests in den Hypothesenmengen abhängig.

Insgesamt bevorzugen wir daher Strukturtests als lokale Hypothesen und definieren die folgende, vorläufige Auswertungsregel:

$$\frac{\texttt{if}(a, b, \texttt{false})}{\texttt{if}(b, a, \texttt{false})}, \quad \text{falls } a \notin \mathcal{A}t^{\texttt{?}}(P) \text{ und } b \in \mathcal{A}t^{\texttt{?}}(P). \tag{8.19}$$

Sind für eine Konjunktion $\texttt{if}(a, b, \texttt{false})$ sowohl a als auch b Strukturtests, so ist es für die symbolische Auswertung der Konjunktion im Allgemeinen vorteilhafter, den Strukturtest mit „weniger“ Prozeduren als lokale Hypothese zu verwenden, da wir so den Strukturtest mit „mehr“ Prozeduren unter einer mächtigeren lokalen Hypothesenmenge auswerten. Wir definieren dazu die folgende Relation:

$$a >_P^{\texttt{?}} b$$

gdw.

$$proc_P(a) \supsetneq proc_P(b) \text{ oder}$$
$$proc_P(a) = proc_P(b) \text{ und } |proc_P(a)| > |proc_P(b)|. \quad {}^{10}$$

Mit Hilfe dieser Relation modifizieren wir Definition (8.19) und definieren die nachfolgende Regel. Hierbei ist zu beachten, dass die Relation $>_P^?$ asymmetrisch ist und daher aufgrund der Regel keine Endlos-Auswertungen möglich sind.

79. Commute then-part atom

$$\frac{\texttt{if}^{\,A}(a, b, \texttt{false})}{\texttt{if}^{\,A}(b, a, \texttt{false})},$$

falls $a \notin \mathcal{A}t^?(P)$ und $b \in \mathcal{A}t^?(P)$ oder $a, b \in \mathcal{A}t^?(P)$ und $a >_P^? b$.

Konjunktionen mit einem negiertem Atom als Konjunktionsglied, d.h. Konjunktionen der Form $\neg a \wedge b$ mit $a, b \in \mathcal{A}t(P)$, können wir durch die `if`-Terme

$$\texttt{if}(a, \texttt{false}, b) \tag{8.20}$$

und

$$\texttt{if}(b, \texttt{if}(a, \texttt{false}, \texttt{true}), \texttt{false}) \tag{8.21}$$

repräsentieren. Es stellt sich auch hier die Frage, welcher der beiden Terme für die symbolische Auswertung besser geeignet ist. Aus den gleichen Gründen wie bei „*Commute then-part atom*“ bevorzugen wir Strukturtests als lokale Hypothesen und definieren die folgende Regel.

80. Commute else-part atom

$$\frac{\texttt{if}^{\,A}(a, \texttt{false}, b)}{\texttt{if}^{\,A}(b, \texttt{if}^{\,A_{\text{std}}}(a, \texttt{false}, \texttt{true}), \texttt{false})},$$

falls $b \in \mathcal{A}t^?(P)$.

Die Regel „*Commute else-part atom*“ vertauscht für $b \in \mathcal{A}t^?(P)$ die Atome a und b. Unabhängig davon, ob a ein Strukturtest ist oder nicht, ist diese Vertauschung sinnvoll. Der Grund hierfür ist, dass das Atom b in Term (8.20) unter der lokalen Hypothese $\neg a$ ausgewertet wird. Ist a ein Strukturtest, so wird b in (8.20) unter dem negierten Strukturtest $\neg a$ ausgewertet. Negierte Strukturtests sind aber deutlich schwächere lokale Hypothesen als Strukturtests, da sie lediglich Disjunktionen von Strukturtests repräsentieren und somit nicht im gleichen Maße wie Strukturtests die Anwendung von Regeln kontrollieren. Wir bevorzugen daher für $b \in \mathcal{A}t^?(P)$ grundsätzlich Term (8.21) zur Repräsentation der Konjunktion $\neg a \wedge b$. Betrachten wir ein Beispiel:

Beispiel 8.12. Sei P das Programm $\langle D_{\texttt{tree}} \rangle$, wobei $D_{\texttt{tree}}$ die Typoperatordefinition aus Abbildung 3.3 bezeichnet. Der Term

$$\texttt{if}(\texttt{?nil}(\texttt{t}), \texttt{false}, \texttt{?node}(\texttt{t}))$$

kann ohne Regel „*Commute else-part atom*“ nicht vereinfacht werden, da der negierte Strukturtest $\neg$`?nil(t)` keine Auswertung des Terms `?node(t)` ermöglicht. Wenden wir jedoch die Regel „*Commute else-part atom*“ an, so erhalten wir den Term

$$\texttt{if}(\texttt{?node}(\texttt{t}), \texttt{if}(\texttt{?nil}(\texttt{t}), \texttt{false}, \texttt{true}), \texttt{false}). \tag{8.22}$$

[10] Hierbei bezeichnet $proc_P(t)$ die Menge der Prozeduren von P, die in t enthalten sind.

Aufgrund der Strukturgleichung `?node(t)` können wir dann für `?nil(t)` die Regel „*Negative hypothesis*“ anwenden und so den Term (8.22) zu

$$\texttt{if}(\texttt{?node}(\texttt{t}), \texttt{if}(\texttt{false}, \texttt{false}, \texttt{true}), \texttt{false})$$

vereinfachen. Die anschließende Anwendung der Regeln „*Keep branch*“ und „*Skip alternatives*“ wertet den Term schließlich zu `?node(t)` aus. □

Wir können die Kommutativität der Konjunktion auch in komplizierteren `if`-Termen nutzen, um Strukturtests als lokale Hypothesen verfügbar zu machen und so die symbolische Auswertung von Termen zu ermöglichen. Betrachten wir dazu ein Beispiel.

Beispiel 8.13. Sei P das Programm $\langle D_{\texttt{list}}, D_{\texttt{ordered}} \rangle$, wobei $D_{\texttt{list}}$ die Typoperatordefinition aus Abbildung 3.1 bezeichnet und $D_{\texttt{ordered}}$ die folgende Prozedurdefinition:

```
function ordered(l : list[nat]) : bool ⇐
  if(?empty(l),
     true,
     if(?empty(tl(l)),
        true,
        if(hd(l)>hd(tl(l)),
           false,
           ordered(tl(k))))).
```

Der Term

$$\texttt{if}(\texttt{ordered}(\texttt{l}), \texttt{true}, \texttt{if}(\texttt{?empty}(\texttt{l}), \texttt{ordered}(\texttt{r}), \texttt{true}))$$

kann mit den bisher definierten Regeln nicht symbolisch ausgewertet werden. Da der Term jedoch die Implikation $\neg\texttt{ordered}(\texttt{l}) \wedge \texttt{?empty}(\texttt{l}) \rightarrow \texttt{ordered}(\texttt{r})$ repräsentiert, können wir den Term durch

$$\texttt{if}(\texttt{?empty}(\texttt{l}), \texttt{if}(\texttt{ordered}(\texttt{l}), \texttt{true}, \texttt{ordered}(\texttt{r})), \texttt{true}) \tag{8.23}$$

ersetzen. Term (8.23) können wir dann durch die folgende symbolische Auswertung zu `true` vereinfachen:

$\mathcal{U} \vdash$ $\texttt{if}(\texttt{?empty}(\texttt{l}), \texttt{if}(\underline{\texttt{ordered}(\texttt{l})}, \texttt{true}, \texttt{ordered}(\texttt{r})), \texttt{true})$ $\rightarrow$

$\texttt{if}(\texttt{?empty}(\texttt{l}), \texttt{if}(\texttt{if}(\underline{\texttt{?empty}(\texttt{l})}, \texttt{true}, \ldots), \texttt{true}, \texttt{ordered}(\texttt{r})), \texttt{true})$ $\rightarrow$

$\texttt{if}(\texttt{?empty}(\texttt{l}), \texttt{if}(\underline{\texttt{if}(\texttt{true}, \texttt{true}, \ldots)}, \texttt{true}, \texttt{ordered}(\texttt{r})), \texttt{true})$ $\rightarrow$

$\texttt{if}(\texttt{?empty}(\texttt{l}), \underline{\texttt{if}(\texttt{true}, \texttt{true}, \texttt{ordered}(\texttt{r}))}, \texttt{true})$ $\rightarrow$

$\underline{\texttt{if}(\texttt{?empty}(\texttt{l}), \texttt{true}, \texttt{true})}$ $\rightarrow$

$\texttt{true}$

□

Um Auswertungen wie in Beispiel 8.13 zu ermöglichen definieren wir die folgenden Regeln:

81. Move then-part atom

$$\frac{\texttt{if}^{\,A}(a, \texttt{if}^{\,B}(b, c, d), e)}{\texttt{if}^{\,B}(b, \texttt{if}^{\,A}(a, c, d), e)},$$

falls $d \simeq_{\mathcal{U}} e$ und

(1) $a \notin \mathcal{At}^{?}(P)$ und $b \in \mathcal{At}^{?}(P)$, oder

(2) $a, b \in \mathcal{At}^{?}(P)$ und $a >_P^{?} b$.

82. Move else-part atom

$$\frac{\texttt{if}^{\,A}(a, b, \texttt{if}^{\,B}(c, d, e))}{\texttt{if}^{\,B}(c, \texttt{if}^{\,A}(a, b, d), e)},$$

falls $b \simeq_{\mathcal{U}} e$ und $c \in \mathcal{At}^{?}(P)$.

Die Korrektheit der Regel lässt sich leicht mittels Wahrheitstafel ermitteln. Für Regel 81 erhalten wir die Wahrheitstafel aus Abbildung 8.5 und für Regel 82 erhalten wir die Wahrheitstafel aus Abbildung 8.6.

8.9 Regeln zum Setzen der A-Umgebung

Die letzten Regeln, die wir in diesem Kapitel definieren möchten, sind die ersten 5 Regeln des Auswertungskalküls. Da es sich bei diesen Regeln um technische Regeln handelt, haben wir die Definition dieser Regeln bis zum Schluss aufgehoben.

Einige der in den folgenden Kapiteln definierten Auswertungsregeln

$$\frac{t}{r}, \quad \text{falls } \Phi(t, r)$$

müssen sicherstellen, dass Teilterme s des Ergebnisterms r in nachfolgenden Auswertungsschritten mit einer festen A-Umgebung $\mathcal{U}_s$ ausgewertet werden. Hierzu werden die Teilterme s mit der A-Umgebung $\mathcal{U}_s$ annotiert:

$$s^{\langle \ldots, \mathcal{U}_s, \ldots \rangle}.$$

Der Term s wird dann zunächst unter der A-Umgebung $\mathcal{U}_s$ ausgewertet. Kann der Term s unter der A-Umgebung $\mathcal{U}_s$ nicht weiter ausgewertet werden, so wird die A-Umgebung $\mathcal{U}_s$ aus der Annotation des Terms s entfernt und der Term wird anschließend unter der aktuellen A-Umgebung $\mathcal{U}$ ausgewertet. Um dieses Verhalten zu realisieren, definieren wir in diesem Abschnitt die folgenden Regeln:

1. *Set e-environment*
2. *Reset e-environment*
3. *Evaluate e-environment argument*
4. *Evaluate e-environment let argument*
5. *Evaluate e-environment let body*

⋮ ⋮

a	b	c	d, e	$\texttt{if}(a, \texttt{if}(b, c, d), e)$	$\texttt{if}(b, \texttt{if}(a, c, d), e)$
true	true	true	true	true	true
true	true	true	false	true	true
true	true	false	true	false	false
true	true	false	false	false	false
true	false	true	true	true	true
true	false	true	false	false	false
true	false	false	true	true	true
true	false	false	false	false	false
false	true	true	true	true	true
false	true	true	false	false	false
false	true	false	true	true	true
false	true	false	false	false	false
false	false	true	true	true	true
false	false	true	false	false	false
false	false	false	true	true	true
false	false	false	false	false	false

Abbildung 8.5: Wahrheitstafel für Regel „*Move then-part atom*“. Es ist zu beachten, dass $d \simeq_{\mathcal{U}} e$ gilt und damit die Wahrheitswerte von d und e immer übereinstimmen.

a	b, e	c	d	$\texttt{if}(a, b, \texttt{if}(c, d, e))$	$\texttt{if}(c, \texttt{if}(a, b, d), e)$
true	true	true	true	true	true
true	true	true	false	true	true
true	true	false	true	true	true
true	true	false	false	true	true
true	false	true	true	false	false
true	false	true	false	false	false
true	false	false	true	false	false
true	false	false	false	false	false
false	true	true	true	true	true
false	true	true	false	false	false
false	true	false	true	true	true
false	true	false	false	true	true
false	false	true	true	true	true
false	false	true	false	false	false
false	false	false	true	false	false
false	false	false	false	false	false

Abbildung 8.6: Wahrheitstafel für Regel „*Move else-part atom*“. Es ist zu beachten, dass $b \simeq_{\mathcal{U}} e$ gilt und damit die Wahrheitswerte von b und e immer übereinstimmen.

Wir definieren zunächst die beiden Regel „*Set e-environment*“ und „*Reset e-environment*“. Zur Definition dieser Regeln benötigen wir die Funktion $\mathbf{env}_{\mathcal{U}}$. Diese Funktion setzt die A-Umgebung der Annotation A des übergebenen Terms t^{A} auf $\mathcal{U}$:

$$\mathbf{env}_{\mathcal{U}}(t^{A}) := t^{A'} \quad \text{mit} \quad \begin{array}{|ll|}\hline A: & \text{Umg.} \\ \hline A': & \mathcal{U} \\ \hline \end{array}$$

Die Regel „*Set e-environment*“ wertet einen Term $t^{\langle \ldots, \mathcal{U}_t, \ldots \rangle}$ unter der $\mathcal{U}_t$ Umgebung aus, falls eine solche Auswertung möglich ist. Die Regel „*Reset e-environment*“ entfernt aus einem Term $t^{\langle \ldots, \mathcal{U}_t, \ldots \rangle}$ die A-Umgebung $\mathcal{U}_t$, falls t unter der A-Umgebung nicht ausgewertet werden kann. Die Regeldefinitionen lauten wie folgt:

1. Set e-environment

$$\frac{t^{A_t}}{t'}, \qquad \begin{array}{|ll|}\hline & \text{Umg.} \\ \hline A_t: & \mathcal{U}_t \\ \hline \end{array}$$

falls $\mathcal{U}_t \neq \bot$, $\mathcal{U}_t \vdash \mathbf{env}_{\bot}(t^{A_t}) \rightarrow r^{A_r}$ und

$$t' = \begin{cases} \mathbf{env}_{\mathcal{U}_t}(r^{A_r}) & \text{falls } \mathcal{U}_r = \bot, \\ r^{A_r} & \text{sonst} \end{cases}$$

gilt. Hierbei bezeichnet A_r die folgende A-Annotation:

$$\begin{array}{|ll|}\hline & \text{Umg.} \\ \hline A_r: & \mathcal{U}_r \\ \hline \end{array}$$

2. Reset e-environment

$$\frac{t^{A_t}}{t^{B_t}}, \qquad \begin{array}{|ll|}\hline & \text{Umg.} \\ \hline A_t: & \mathcal{U}_t \\ B_t: & \bot \\ \hline \end{array}$$

falls $\mathcal{U}_t \neq \bot$ gilt.

Für die Regel „*Set e-environment*“ sind die folgenden Punkte zu beachten:

- Die Umgebung $\mathcal{U}_t$ ist während der Auswertung von t aus der Annotation A zu entfernen, da die Regel ansonsten mit sich selbst in eine Endlosschleife gerät. Zum Entfernen der A-Umgebung verwenden wir die Funktion $\mathbf{env}_{\bot}$.
- Die Fallunterscheidung zur Festlegung von t' ist notwendig, um zu verhindern, dass eine entsprechende A-Umgebung in A_r überschrieben wird.

Um zu gewährleisten, dass der Term $s^{\mathcal{U}_s}$ mit der A-Umgebung $\mathcal{U}_s$ ausgewertet wird, ist es notwendig, weitere Regeln zu definieren.[11] Nehmen wir beispielsweise an, der Term s hätte die Form $cons(s')$, wobei $cons$ einen Konstruktor bezeichnet. Nehmen wir weiter an s kann mit Hilfe der A-Umgebung $\mathcal{U}_s$ zu einem Term q

[11] Im Folgenden notieren wir der Übersichtlichkeit halber lediglich die A-Umgebung der A-Annotationen.

ausgewertet werden. Wir möchten nun gewährleisten, dass der Term $cons^{\mathcal{U}_s}(s')$ unabhängig vom Kontext, in dem der Term enthalten ist, zu q ausgewertet wird. Bezeichne *sel* einen Selektor des Konstruktors *cons* so ist dies jedoch für den Kontext $sel(\ldots)$ nicht der Fall:

$$\mathcal{U} \vdash sel(cons^{\mathcal{U}_s}(s')) \to s'.$$

Um solche Auswertungen zu vermeiden, definieren wir die folgenden Regeln:

3. Evaluate e-environment argument

$$\frac{f^{A}(t_1, \ldots, t_i, \ldots, t_n)}{f^{A}(t_1, \ldots, t_i', \ldots, t_n)},$$

falls $\mathcal{U} \vdash t_i \to t_i'$ und t_i einen Teilterm enthält, der mit einer A-Umgebung annotiert ist. Weiter dürfen für alle $j \in \{1, \ldots i-1\}$ die Terme t_j keine Teilterme enthalten, die mit einer A-Umgebung annotiert sind.

4. Evaluate e-environment let argument

$$\frac{\texttt{let}^{A}\ x\texttt{:=}t\ \texttt{in}\ r\ \texttt{end}}{\texttt{let}^{A}\ x\texttt{:=}t'\ \texttt{in}\ r\ \texttt{end}},$$

falls $\mathcal{U} \vdash t \to t'$ und t einen Teilterm enthält, der mit einer A-Umgebung annotiert ist.

5. Evaluate e-environment let body

$$\frac{\texttt{let}^{A}\ x\texttt{:=}t\ \texttt{in}\ r\ \texttt{end}}{\texttt{let}^{A}\ x\texttt{:=}t\ \texttt{in}\ r'\ \texttt{end}},$$

falls $\mathcal{U} \vdash r \to r'$ und r einen Teilterm enthält, der mit einer A-Umgebung annotiert ist.

Die Regeln gewährleisten, dass für einen Term zunächst alle Teilterme ausgewertet werden, für die eine A-Umgebung in ihrer Annotation enthalten ist. Erst wenn diese Teilterme vollständig ausgewertet sind, wir der restliche Term ausgewertet.

Anmerkung 8.14. Die Notwendigkeit der in diesem Abschnitt definierten Regeln ergibt sich aus den in den Kapitel 10-13 definierten „*Unfold*“-, „*Assumption*“- und „*Functionality*“-Regeln. Für einen konkreten Anwendungsfall der Regeln verweisen wir auf Abschnitt F.7 in Anhang F. □

Kapitel 9

Die „*Execute*“-Regeln

Wir beschäftigen uns in diesem Kapitel mit einem der zentralen Probleme des Auswertungskalküls: Der Auswertung von Prozeduraufrufen. Für eine erfolgreiche symbolische Auswertung der Goal-Terme ist es von entscheidender Bedeutung, *welche* Prozeduraufrufe und *wie weit* diese Prozeduraufrufe ausgewertet werden sollen. So muss beispielsweise für den Induktionsschritt aus Beispiel 8.9 der Prozeduraufruf `plus(x, 0)` des Goal-Terms genau bis zum rekursiven Aufruf `succ(plus(pred(x), 0))` ausgewertet werden, damit die Induktionshypothese angewendet werden kann. Würde der Prozeduraufruf gar nicht oder zu weit ausgewertet werden, so wäre die Induktionshypothese nicht anwendbar und die symbolische Auswertung könnte den Goal-Term nicht zu `true` vereinfachen.

Zahlreiche Fallstudien mussten gerechnet und analysiert werden, um geeignete Heuristiken zur Auswertung von Prozeduraufrufen zu finden. Die gefundenen Heuristiken werden im Wesentlichen durch die „*Execute*“-Regeln realisiert. Diese Regeln wollen wir in diesem Kapitel definieren. Hierzu gehen wir wie folgt vor: Zunächst betrachten wir rekursiv-definierte Prozeduren. Für diese Prozeduren definieren wir in den Abschnitten 9.1-9.3 Regeln zur Auswertung der folgenden Arten von Prozeduraufrufen:

- Prozeduraufrufe mit Konstruktorgrundtermen als Argumente.
- Prozeduraufrufe, die bei Auswertung zu einem rekursiven Fall führen.
- Prozeduraufrufe, die bei Auswertung zu einem Basisfall führen.

Anschließend definieren wir in Abschnitt 9.4 Regeln zur Auswertung von Prozeduraufrufen von nicht-rekursive-definierten Prozeduren. Insgesamt führen wir so in den Abschnitten 9.1-9.4 die folgenden Auswertungsregeln ein:

⋮ ⋮
41. Execute procedure call (constructor ground terms)
43. Execute procedure call (recursive cases)
45. Execute procedure call (non recursive cases)
47. Execute procedure call (no additional case analysis)
51. Execute procedure call (additional case analysis)
⋮ ⋮

Für diese Auswertungsregeln definieren wir in Abschnitt 9.5 kommutierte Versionen, um für kommutative Prozeduren zusätzliche symbolische Auswertungen zu ermöglichen. Wir schließen das Kapitel in Abschnitt Abschnitt 9.6 mit einigen Anmerkungen zur Implementierung der Regeln im ✓`eriFun`-System.

Jede der in diesem Kapitel definierten Regeln wird nach der Regel „*Evaluate argument*“ (siehe Abschnitt 8.7) angewendet. Das bedeutet, dass Prozeduraufrufe durch die „*Execute*“-Regeln erst dann ausgewertet werden, wenn die Argumente der Prozeduraufrufe vollständig ausgewertet sind. Weiter sind alle Regeln auf Basis des folgenden Regelschemas definiert:

$$\frac{f(t_1,\ldots,t_n)}{\sigma_{\xi_{f(t_1,\ldots,t_n)}}(R_f^*)}, \qquad \begin{array}{l}\text{falls } f \in \Sigma,\ \sigma = \{x_1/t_1,\ldots,x_n/t_n\}\\ \text{und } \ldots\end{array} \tag{9.1}$$

Die Typsubstitution $\xi_{f(t_1,\ldots,t_n)}$ bezeichnet hierbei die auf Seite 31 für beliebige Funktionsaufrufe t eingeführte Typsubstitution ξ_t. Die Anwendungsbedingung $f \in \Sigma$ gewährtleistet, dass der Prozeduraufruf $f(t_1,\ldots,t_n)$ nur dann ausgewertet wird, wenn f in der Signatur der auszuwertenden Prozeduren Σ enthalten ist.

9.1 Konstruktorgrundterme als Argumente

Für eine rekursiv-definierte Prozedur f bezeichnen wir einen Prozeduraufruf $f(t_1, \ldots, t_n)$ als *trivial*, falls die Argumente der Induktionsvariablen einer Relationenbeschreibung $R \in \mathit{grds}(f)$ Konstruktorgrundterme sind. Triviale Prozeduraufrufe rekursiv-definierter Prozeduren können meist *rekursionsvollständig* auswertet werden, d.h. bei der Auswertung des instantiierten Prozedurrumpfs $R_f^*[x_1/t_1, \ldots, x_n/t_n]$ verschwinden alle rekursiven Aufrufe von f. Die in diesem Abschnitt definierte Regel „*Execute procedure call (constructor ground terms)*“ implementiert daher die folgende Heuristik:

Heuristik (Auswertung trivialer Prozeduraufrufe):
Triviale Prozeduraufrufe $f(t^*)$ einer rekursiv-definierten Prozedur f werden immer ausgewertet.

Betrachten wir dazu ein Beispiel:

Beispiel 9.1. Sei P das Programm $\langle D_{\texttt{plus}}\rangle$, wobei $D_{\texttt{plus}}$ die entsprechende Prozedurdefinition aus Abbildung 8.2 bezeichnet. Sei weiter die Menge $\mathit{grds}(\texttt{plus})$ der generalisierten Relationenbeschreibungen von `plus` durch $\{R\}$ gegeben, wobei gilt

$$R = \{\langle\{\neg\texttt{?0(x)}\}, \{\{\texttt{x/pred(x)}\}\}\rangle\}\,.$$

Dann ist $\{\texttt{x}\}$ die Menge der Induktionsvariablen der Relationenbeschreibung R. Mit Hilfe der Heuristik können wir dann den Prozeduraufruf

$$\texttt{plus(1, plus(x, y))}$$

rekursionsvollständig auswerten und zu $\texttt{succ(plus(x, y))}$ vereinfachen:

$$\begin{array}{lll}
\mathcal{U} \vdash & \underline{\texttt{plus(1, plus(x, y))}} & \rightarrow\\
& \texttt{if(}\underline{\texttt{1=0}}\texttt{, plus(x, y), succ(plus(pred(1), plus(x, y))))} & \rightarrow\\
& \underline{\texttt{if(false, plus(x, y), succ(plus(pred(1), plus(x, y))))}} & \rightarrow\\
& \texttt{succ(plus(}\underline{\texttt{pred(1)}}\texttt{, plus(x, y)))} & \rightarrow\\
& \texttt{succ(}\underline{\texttt{plus(0, plus(x, y))}}\texttt{)} & \rightarrow\\
& \texttt{succ(if(}\underline{\texttt{0=0}}\texttt{, plus(x, y), succ(plus(pred(0), plus(x, y)))))} & \rightarrow\\
& \texttt{succ(}\underline{\texttt{if(true, plus(x, y), succ(plus(pred(0), plus(x, y))))}}\texttt{)} & \rightarrow\\
& \texttt{succ(plus(x, y))}\,. &
\end{array}$$

□

Für unvollständig-definierte Prozeduren f müssen wir sicherstellen, dass nur solche trivialen Prozeduraufrufe $f(t_1, \ldots, t_n)$ ausgewertet werden, deren Auswertung unabhängig von den rekursiven Aufrufen von f ist, die durch das Symbol $*$ in den Prozedurrumpf R_f^* eingefügt wurden. Hierzu definieren wir den *Exception-Guard*:

Definition 9.2 (Exception-Guard)**.** Sei P ein Programm und $D \in P$ eine Prozedurdefinition der Form

$$\texttt{function}\ f(x_1 : \tau_1, \ldots, x_n : \tau_n) : \tau \ \Leftarrow\ R_f.$$

Der *Exception-Guard* der Prozedur f bzgl. P ist dann definiert als

$$except_f^P \quad := \quad OR\left(\bigcup_{\pi \in \mathcal{P}os^*(R_f)} AND\,(cond(\pi, R_f))\right).$$

□

Ist P aus dem Zusammenhang klar oder irrelevant, so schreiben wir für den Exception-Guard auch kurz $except_f$. Der Exception-Guard $except_f[x_1/t_1, \ldots, x_n/t_n]$ ist genau dann ungültig, wenn die Auswertung von $f(t_1, \ldots, t_n)$ unabhängig von den Symbolen $*$ im Prozedurrumpf R_f ist. Wir werten daher einen trivialen Prozeduraufruf $f(t_1, \ldots, t_n)$ nur dann aus, falls der Term $except_f[x_1/t_1, \ldots, x_n/t_n]$ zu `false` vereinfacht werden kann. Zur Überprüfung dieser Bedingung definieren wir das folgende Prädikat:

$$det_{\mathcal{U}}(f(t_1, \ldots, t_n)) \quad :\Longleftrightarrow \quad \mathcal{U} \vdash \mathbf{a}_{A_{\text{top}}}(except_f[x_1/t_1, \ldots, x_n/t_n]) \rightarrow^! \texttt{false}^{1}$$

Wir definieren dann die Regel „*Execute procedure call (constructor ground terms)*“ wie folgt: [2]

41. Execute procedure call (constructor ground terms)

$$\frac{f^{\,A}(t_1, \ldots, t_n)}{\sigma_{\xi_{f(\ldots)}}(\mathbf{s}_f^*(\mathbf{un}_f^{s,u,0}(R_f)))},$$

	S-Lim.	U-Lim.
A:	s	u

falls $f \in \Sigma \cap \Sigma^{\text{rec}}(P)$, $\sigma = \{x_1/\mathbf{a}_{A_0}(t_1), \ldots, x_n/\mathbf{a}_{A_0}(t_n)\}$ gilt und eine Relationenbeschreibung $R \in grds_P(f)$ existiert mit $\mathbf{e}(\sigma(x)) \in \mathcal{T}erm^{\text{cons}}(P)$ für alle $x \in iv(R)$. Weiter muss gelten $det_{\mathcal{U}}(f(\mathbf{e}(t_1), \ldots, \mathbf{e}(t_n))))$.

Anmerkung 9.3. Nicht in allen Fällen können triviale Prozeduraufrufe rekursionsvollständig ausgewertet werden (siehe Beispiel F.1 in Anhang F). Weiter ist es möglich, dass durch die Auswertung trivialer Prozeduraufrufe Endlos-Auswertungen entstehen (siehe Beispiel F.2 in Anhang F). Um diese Probleme zu beseitigen, müsste die Auswertung trivialer Prozeduraufrufe auf die Fälle beschränkt werden, in denen die Argumente der *relevanten Variablen* einer Relationenbeschreibung $R \in grds(f)$

[1] Aufgrund des Search-Limits der A-Annotation A_{top} werden während der symbolischen Auswertung des Exception-Guards keine Quantifikationen und auch keine „*Functionality*“-Regeln angewendet. Das Label $\top$ der A-Annotation A_{top} sperrt, wie wir bereits in Abschnitt 6.1 beschrieben haben, die Auswertung von Prozeduraufrufen. Ausgenommen hiervon sind triviale Prozeduraufrufe. Diese Einschränkungen der Auswertung des Exception-Guards sind notwendig, um eine effiziente und trotzdem ausreichend mächtige Überprüfung des Exception-Guards zu ermöglichen.

[2] Zum Annotieren des Prozedurrumpfs R_f verwenden wir die Funktionen $\mathbf{un}_f^{s,u,0}$ und $\mathbf{s}_f^*$. Weiter annotieren wir die Argumente t_i mit der A-Annotation A_0. Die Notwendigkeit dieser Annotationen erklären wir in den abschließenden Anmerkungen am Schuss dieses Kapitels. Für den Moment ignorieren wir die A-Annotationen.

Konstruktorgrundterme sind. Diese Beschränkung verhindert jedoch in einigen Fallstudien notwendige Auswertungen von Prozeduraufrufen (siehe Beispiel F.3 in Anhang F). Da außerdem in keiner unserer Fallstudien die beschriebenen Probleme real auftraten, verzichten wir auf eine Beschränkung der Auswertung trivialer Prozeduraufrufe. □

Anmerkung 9.4. Die Regel „*Execute procedure call (constructor ground terms)*" ist die einzige Regel zur Auswertung von Prozeduraufrufen rekursiv-definierter Prozeduren, die nicht durch die Kontextmenge C gesperrt wird. Die Gründe hierfür sind, dass erstens die Anwendungsbedingungen der Regel sehr effizient überprüft werden können und zweitens die rekursionsvollständige Auswertung eines Prozeduraufrufs $f(t_1, \ldots, t_n)$ unabhängig vom Kontext der Auswertung immer sinnvoll ist. □

9.2 Auswertung zu rekursiven Fällen

Typischerweise wird die Gültigkeit eines Induktionsschritts

$$H, IH \vdash g$$

dadurch bewiesen, dass der Goal-Term g auf eine oder mehrere der Induktionshypothesen aus IH zurückgeführt wird. Die Induktionshypothesen und der Goal-Term weisen hierbei syntaktische Unterschiede auf. Es ist die Aufgabe der symbolischen Auswertung, diese syntaktischen Unterschiede zu entfernen und so die Anwendung der Induktionshypothesen zu ermöglichen. Wir definieren hierzu in diesem Abschnitt die Regel „*Execute procedure call (recursive-cases)*". Diese Regel ersetzt Prozeduraufrufe in einem Goal-Term durch den instantiierten Rumpf, falls der instantiierte Rumpf sich anschließend durch symbolische Auswertung auf die rekursiven Aufrufe der Prozedur vereinfachen lässt. Dadurch reduziert sich der syntaktische Unterschied zwischen Goal-Term und Induktionshypothesen. Die Regel „*Execute procedure call (recursive-cases)*" ist somit eine der zentrale Regeln des Auswertungskalküls. Zur Definition der Regel gehen wir wie folgt vor. Wir definieren zunächst in Abschnitt 9.2.1 die Heuristik, auf deren Basis die Regel „*Execute procedure call (recursive-cases)*" definiert ist. Anschließend gehen wir in den Abschnitten 9.2.2 - 9.2.5 auf verschiedene Aspekte ein, die bei der Realisierung der Heuristik durch die Regel zu beachtet sind. Darauf aufbauend definieren wir schließlich in Abschnitt 9.2.6 die Auswertungsregel.

9.2.1 Heuristik zur Auswertung

Um die durch die Regel „*Execute procedure call (recursive-cases)*" realisierte Heuristik zu motivieren, betrachten wir ein Beispiel:

Beispiel 9.5. Sei das Programm P gegeben durch $\langle \ldots, D_{\texttt{null}}, \ldots \rangle$, wobei $D_{\texttt{null}}$ die folgende Prozedurdefinition bezeichnet:

$$\begin{aligned} D_{\texttt{null}} \;=\; & \texttt{function null(x : nat) : bool} \Leftarrow \\ & \quad \texttt{if}(a, \texttt{0}, \\ & \qquad \texttt{if}(b, \texttt{0}, \\ & \qquad\quad \texttt{if}(c, \\ & \qquad\qquad \texttt{null}(C_1[x]), \\ & \qquad\qquad \texttt{null}(C_2[x])))) \,. \end{aligned}$$

Die Terme a, b, c sowie die Kontexte C_1, C_2 sein so gewählt, dass die Relationenbeschreibung $R = \{R_1, R_2\}$ mit

$$\begin{aligned} R_1 &= \langle \{\neg b, c\}, \{x / C_1[x]\} \rangle, \\ R_2 &= \langle \{\neg b, \neg c\}, \{x / C_2[x]\} \rangle \end{aligned}$$

fundiert ist. Die Menge *grds*(`null`) der generalisierten Relationenbeschreibungen von `null` sei dann gegeben durch $\{R\}$. Aus der Fundiertheit der Relationenbeschreibung R folgt, dass die Prozedur `null` terminiert. Hierbei gilt:

(1) Für die Terminierung des ersten rekursiven Aufrufs $f(C_1[x])$ ist die Gültigkeit der Konjunktion $guard_1 = AND(\neg b, c)$ verantwortlich.

(2) Für die Terminierung des zweiten rekursiven Aufrufs $f(C_2[x])$ ist die Gültigkeit der Konjunktion $guard_2 = AND(\neg b, \neg c)$ verantwortlich.

Wir bezeichnen die Konjunktionen $guard_1$ und $guard_2$ als die *Recursion-Guards* der entsprechenden rekursiven Aufrufe. Können wir für einen Prozeduraufruf `null`(t) den instantiierten Recursion-Guard $guard_1[\mathtt{x}/t]$ bzw. $guard_2[\mathtt{x}/t]$ zu `true` auswerten, so können wir den Prozeduraufruf auf den entsprechenden rekursiven Aufruf zurückführen. Diese Eigenschaft der Recursion-Guards wollen wir zur Auswertung von Prozeduraufrufen nutzen, um die Unterschiede zwischen Induktionshypothesen und Goal-Term zu reduzieren. Betrachten wir hierzu den Beweis des folgenden Lemmas:

```
lemma null_is_zero ⇐ all x, y : nat
  null(x)=0 .
```

Zum Beweis der Gültigkeit des Lemmas schlägt der Prozeduraufruf `null(x)` eine Induktion über R vor.[3] Für diese Induktion erhalten wir die folgenden Sequenzen als Induktionsschritte:

$$\{\neg b, c\}, \{\mathtt{null}(C_1[\mathtt{x}])\mathtt{=0}\} \quad \vdash \quad \mathtt{null(x)=0}, \tag{9.2}$$

$$\{\neg b, \neg c\}, \{\mathtt{null}(C_2[\mathtt{x}])\mathtt{=0}\} \quad \vdash \quad \mathtt{null(x)=0}. \tag{9.3}$$

Jede atomare Relationenbeschreibung von R definiert einen Induktionsschritt. Da die atomaren Relationenbeschreibungen auf Basis der rekursiven Aufrufe definiert sind, ist somit für jeden rekursiven Aufruf von `null` ein Induktionsschritt definiert. Die Hypothesen der Induktionsschritte repräsentieren dabei den Recursion-Guard des rekursiven Aufrufs `null`$(C_i[x])$ für den der Induktionsschritt definiert wurde. Auf diesen rekursiven Aufruf `null`$(C_i[x])$ müssen wir dann den Prozeduraufruf von `null` im Goal-Term zurückführen, um die Induktionshypothese anwenden zu können. Wir ersetzen daher während der symbolischen Auswertung einen Prozeduraufruf `null`(t) durch den instantiierten Rumpf, falls aufgrund der Hypothesen der Sequenz ein Recursion-Guard der Prozedur `null` gültig ist. Die Hypothesen des Induktionsschritts sind während der symbolischen Auswertung als globale Hypothesen der A-Umgebung $\mathcal{U}$ verfügbar:

$$\begin{aligned}
\mathcal{U} \;\vdash\; & \underline{\mathtt{null(x)=0}} && \rightarrow \\
& \mathtt{if}(a, \mathtt{0}, \mathtt{if}(\underline{b}, \mathtt{0}, \mathtt{if}(c, \mathtt{null}(C_1[x]), \mathtt{null}(C_2[x]))))\mathtt{=0} && \rightarrow \\
& \mathtt{if}(a, \mathtt{0}, \underline{\mathtt{if}(\mathtt{false}, \mathtt{0}, \mathtt{if}(c, \mathtt{null}(C_1[x]), \mathtt{null}(C_2[x])))})\mathtt{=0} && \rightarrow \\
& \mathtt{if}(a, \mathtt{0}, \mathtt{if}(\underline{c}, \mathtt{null}(C_1[x]), \mathtt{null}(C_2[x])))\mathtt{=0} && \rightarrow \\
& \mathtt{if}(a, \mathtt{0}, \underline{\mathtt{if}(\mathtt{true}, \mathtt{null}(C_1[x]), \mathtt{null}(C_2[x]))})\mathtt{=0} && \rightarrow \\
& \underline{\mathtt{if}(a, \mathtt{0}, \mathtt{null}(C_1[x]))\mathtt{=0}} && \rightarrow
\end{aligned}$$

[3] Normalerweise wird für die Induktion eine der generalisierten Relationenbeschreibungen R einer rekursiv-definierten Prozedur f verwendet, für die im Goal-Term ein Prozeduraufruf $f(t_1, \ldots, t_n)$ existiert, so dass die Argumente der Induktionsvariablen der Relationenbeschreibung in diesem Prozeduraufruf durch verschiedene Variablen gegeben sind. Wir sprechen dann davon, dass der Prozeduraufruf $f(t_1, \ldots, t_n)$ die Relationenbeschreibung R zur Induktion vorschlägt. Dieses Verfahren zur Auswahl einer Relationenbeschreibung für die Induktion wird in der Literatur „*Recursion Analysis*“ oder „*Induction Heuristic*“ genannt [11, 47].

$$\texttt{if}(a, \underline{\texttt{0=0}}, \texttt{null}(C_1[x])\texttt{=0}) \quad \rightarrow$$
$$\texttt{if}(a, \texttt{true}, \texttt{null}(C_1[x])\texttt{=0})\ .$$

Für den Prozeduraufruf `null(x)` ist der instantiierte Recursion-Guard $AND(\neg b, c)$ aufgrund der globalen Hypothesen der A-Umgebung $\mathcal{U}$ gültig. Wir werten daher im ersten Auswertungsschritt diesen Prozeduraufruf aus. Für den Prozeduraufruf $\texttt{null}(C_1[x])$ ist hingegen kein instantiierter Recursion-Guard gültig. Wir werten diesen Prozeduraufruf daher nicht weiter aus. Wir haben damit den Goal-Term des Induktionsschritts (9.2) auf eine Konsequenz der Induktionshypothese reduziert und so die Gültigkeit des Induktionsschritts bewiesen. □

Allgemein ersetzen wir während der symbolischen Auswertung einen Prozeduraufruf $f(t_1, \ldots, t_n)$ durch seinen instantiierten Rumpf, falls ein instantiierter Recursion-Guard der Prozedur f aufgrund der globalen Hypothesen zu `true` ausgewertet werden kann. Die *Recursion-Guards* einer Prozedur f sind hierbei wie folgt definiert:

Definition 9.6 (Recursion-Guards)**.** Sei P ein terminierendes Programm und $f \in \Sigma^{\text{rec}}(P)$ eine rekursiv-definierte Prozedur. Die Menge *rec-guards*$_P(f)$ der *Recursion-Guards* von f ist definiert durch

$$\textit{rec-guards}_P(f) := \left\{ AND(D) \mid \langle D, \Delta\rangle \in R \text{ mit } R \in \bigcup \textit{grds}_P(f) \right\}.$$

Ist P aus dem Zusammenhang klar, so schreiben wir für *rec-guards*$_P(f)$ auch kurz *rec-guards*(f). □

Die durch die Regel „*Execute procedure call (recursive cases)*“ realisierte Heuristik lässt sich dann wie folgt formulieren:

Heuristik (Auswertung zur rekursiven Fällen):
Ein Prozeduraufruf $f(t^*)$ einer rekursiv-definierten Prozedur f wird ausgewertet, falls *ein* instantiierter Recursion-Guard $\textit{guard}[x^*/t^*]$ mit $\textit{guard} \in \textit{rec-guards}(f)$ aufgrund der globalen Hypothesen zu `true` ausgewertet werden kann.

Die Heuristik stellt sicher, dass durch die Auswertung der Prozeduraufrufe die syntaktischen Unterschiede zwischen Induktionshypothesen und Goal-Term herausgeschoben werden. Dieses Herausschieben wird allgemein als „*Rippling-Out*“ bezeichnet [4, 13]. Betrachten wir dazu ein Beispiel:

Beispiel 9.7. Sei P das Programm $\langle D_{\texttt{dbl}}, D_{\texttt{half}}\rangle$, wobei $D_{\texttt{dbl}}$ und $D_{\texttt{half}}$ die entsprechenden Prozedurdefinitionen aus Abbildung 8.2 bezeichnen. Seien weiter die Mengen der generalisierten Relationenbeschreibungen der Prozeduren gegeben durch $\textit{grds}(\texttt{dbl}) = \{R_1\}$ und $\textit{grds}(\texttt{half}) = \{R_2\}$, wobei gilt

$$\begin{aligned} R_1 &= \{\{\langle \neg\texttt{?0(x)}\}, \{\{\texttt{x/pred(x)}\}\}\rangle\}, \\ R_2 &= \{\{\langle \neg\texttt{?0(x)}, \neg\texttt{?0(pred(x))}\}, \{\{\texttt{x/pred(pred(x))}\}\}\rangle\}. \end{aligned}$$

Die Mengen der Recursion-Guards der Prozeduren ist dann gegeben durch

$$\begin{aligned} \textit{rec-guards}(\texttt{dbl}) &= \{\neg\texttt{?0(x)}\}, \\ \textit{rec-guards}(\texttt{half}) &= \{AND(\neg\texttt{?0(x)}, \neg\texttt{?0(pred(x))})\}. \end{aligned}$$

Wir wollen nun das folgende Lemma beweisen:

$$\begin{array}{l}\texttt{lemma half\_dbl} \Leftarrow \texttt{all x : nat}\\ \quad \texttt{half(dbl(x))=x.}\end{array}$$

Zum Beweis dieses Lemmas ist es notwendig, die Gültigkeit der Sequenz

$$\emptyset, \emptyset \vdash \texttt{half(dbl(x))=x} \tag{9.4}$$

nachzuweisen. Der Teilterm `dbl(x)` schlägt hierzu eine Induktion über die Relationenbeschreibung R_1 vor. Für diese Induktion erhalten wir als Induktionsschritt die folgende Sequenz:

$$\{\neg\texttt{?0(x)}\}, \{\texttt{half(dbl(pred(x)))=x}\} \vdash \texttt{half(dbl(x))=x}\,. \tag{9.5}$$

Es ergibt sich dann folgende symbolische Auswertung des Goal-Terms:

$$\begin{array}{lll}
\mathcal{U} \vdash & \texttt{half(}\underline{\texttt{dbl(x)}}\texttt{)=x} & \overset{(1)}{\rightarrow}\\
& \texttt{half(if(}\underline{\texttt{?0(x)}}\texttt{,0,succ(succ(dbl(pred(x)))))=x} & \overset{(2)}{\rightarrow}\\
& \texttt{half(}\underline{\texttt{if(false,0,succ(succ(dbl(pred(x)))))}}\texttt{)=x} & \overset{(3)}{\rightarrow}\\
& \underline{\texttt{half(succ(succ(dbl(pred(x)))))}}\texttt{=x} & \overset{(4)}{\rightarrow}\\
& \begin{array}{l}\texttt{if(}\underline{\texttt{?0(succ(succ(dbl(pred(x)))))}}\texttt{,}\\ \quad \texttt{0,}\\ \quad \texttt{if(?0(pred(succ(succ(dbl(pred(x))))))),}\\ \qquad \texttt{0,}\\ \qquad \texttt{succ(half(pred(pred(succ(succ(dbl(pred(x))))))))=x}\end{array} & \overset{(5)}{\rightarrow}\\
& \underline{\texttt{if(false,0,...)=x}} & \overset{(6)}{\rightarrow}\\
& \texttt{if(?0(}\underline{\texttt{pred(succ(succ(dbl(pred(x)))))}}\texttt{),0,...)=x} & \overset{(7)}{\rightarrow}\\
& \texttt{if(}\underline{\texttt{?0(succ(dbl(pred(x))))}}\texttt{,0,...)=x} & \overset{(8)}{\rightarrow}\\
& \underline{\texttt{if(false,0,succ(half(pred(pred(succ(succ(dbl(pred(x))))))))=x}} & \overset{(9)}{\rightarrow}\\
& \texttt{succ(half(pred(}\underline{\texttt{pred(succ(succ(dbl(pred(x)))))}}\texttt{)))=x} & \overset{(10)}{\rightarrow}\\
& \texttt{succ(half(}\underline{\texttt{pred(succ(dbl(pred(x))))}}\texttt{))=x} & \overset{(11)}{\rightarrow}\\
& \underline{\texttt{succ(half(dbl(pred(x))))=x}} & \overset{(12)}{\rightarrow}\\
& \underline{\texttt{succ(half(dbl(pred(x))))=succ(pred(x))}} & \overset{(13)}{\rightarrow}\\
& \texttt{half(dbl(pred(x))=pred(x).} &
\end{array}$$

Für den Prozeduraufruf `dbl(x)` ist der instantiierte Recursion-Guard $\neg$`?0(x)` aufgrund der gleich lautenden globalen Hypothese gültig. Wir werten daher im ersten Auswertungsschritt den Prozeduraufruf aus. In den beiden nachfolgenden Auswertungsschritten (2) und (3) kann der Prozedurrumpf dann zu folgendem Term vereinfacht werden:

$$\texttt{half(succ(succ(dbl(pred(x)))))=x.} \tag{9.6}$$

Die Auswertung des Prozeduraufrufs `dbl(x)` hat also einerseits den Unterschied zwischen Goal-Term und Induktionshypothese verringert, da der Term `pred(x)` in den Prozeduraufruf von `dbl` eingefügt wurde. Andererseits hat die Auswertung einen

neuen Kontext um den Prozeduraufruf eingefügt und damit einen neuen Unterschied erzeugt. Dieser neue Unterschied kann nun in den Schritten (4)-(11) durch die Auswertung des Prozeduraufrufs von `half` herausgeschoben und in den Schritten (12)-(13) eliminiert werden. Dadurch haben wir den Goal-Term vollständig auf die Induktionshypothese reduziert. Von entscheidender Bedeutung ist hierbei, dass unsere Heuristik die Auswertung des Prozeduraufrufs

$$\texttt{half}(\texttt{succ}(\texttt{succ}(\texttt{dbl}(\texttt{pred}(\texttt{x})))))$$

im Auswertungsschritt (4) ermöglicht. Grund hierfür ist, dass der instantiierte Recursion-Guard

$$\mathit{AND}(\neg\texttt{?0}(\texttt{x}), \neg\texttt{?0}(\texttt{pred}(\texttt{x})))[\texttt{x}/\texttt{succ}(\texttt{succ}(\texttt{dbl}(\texttt{pred}(\texttt{x}))))]$$

zu `true` ausgewertet werden kann. □

Für die Auswertung der instantiierten Recursion-Guards muss die Verwendung lokaler Hypothesen explizit verboten werden, da ansonsten Endlos-Auswertungen nicht ausgeschlossen werden können (siehe Beispiel F.4 in Anhang F). Dies ist der Grund, warum wir lokale und globale Hypothesen unterscheiden. Außerdem können Endlos-Auswertungen auftreten, wenn im Recursion-Guard $\mathit{guard} \in \mathit{rec\text{-}guards}(f)$ einer Prozedur f die Prozedur selbst enthalten ist (siehe Beispiel F.5 in Anhang F). Wir müssen daher bei der Auswertung der instantiierten Recursion-Guards die Auswertung der Prozedur f verbieten. Wir stellen somit die folgende Anforderung an die symbolische Auswertung der Recursion-Guards:

Anforderung (Auswertung der Recursion-Guards):
Für die Auswertung der instantiierten Recursion-Guards $\mathit{guard}[x^*/t^*]$ ist die Verwendung der lokalen Hypothesen und die Auswertung der Prozedur f verboten.

Wie die Recursion-Guards zur symbolischen Auswertung annotiert werden, besprechen wir in Abschnitt 9.2.4.

9.2.2 Labels in Prozedurrümpfen

Wir besprechen nun wie der durch die Auswertung eines Prozeduraufrufs $f(t^*)$ eingeführte Prozedurrumpf $R_f^*[x^*/t^*]$ zu annotieren ist. Wir konzentrieren uns hierzu zunächst auf die Argumente der rekursiven Aufrufe im Rumpf. Betrachten wir dazu ein Beispiel:

Beispiel 9.8. Sei P das Programm $\langle D_{\texttt{minus}}, D_{\texttt{rem}}\rangle$, wobei $D_{\texttt{minus}}$ und $D_{\texttt{rem}}$ die entsprechenden Prozedurdefinitionen aus Abbildung 9.1 bezeichnen. Seien weiter die Mengen der generalisierten Relationenbeschreibungen der Prozeduren durch $\mathit{grds}(\texttt{minus}) = \{R_{\texttt{x}}, R_{\texttt{y}}\}$ und $\mathit{grds}(\texttt{rem}) = \{R\}$ gegeben, wobei gilt

$$\begin{aligned} R_{\texttt{x}} &= \{\langle\{\neg\texttt{?0}(\texttt{x})\}, \{\{\texttt{x}/\texttt{pred}(\texttt{x})\}\}\rangle\}, \\ R_{\texttt{y}} &= \{\langle\{\neg\texttt{?0}(\texttt{y})\}, \{\{\texttt{y}/\texttt{pred}(\texttt{y})\}\}\rangle\}, \\ R &= \{\langle\{\neg\texttt{?0}(\texttt{x}), \neg\texttt{?0}(\texttt{y})\}, \{\{\texttt{x}/\texttt{minus}(\texttt{x},\texttt{y})\}\}\rangle\}. \end{aligned}$$

Wir wollen nun das folgende Lemma beweisen:

$$\begin{aligned} &\texttt{lemma rem\_one\_is\_zero} \Leftarrow \texttt{x},\texttt{y} : \texttt{nat} \\ &\quad \texttt{if}(\texttt{?0}(\texttt{y}), \texttt{true}, \texttt{if}(\texttt{?0}(\texttt{pred}(\texttt{y})), \texttt{?0}(\texttt{rem}(\texttt{x},\texttt{y})), \texttt{true})). \end{aligned}$$

$$
\begin{array}{rcl}
D_{\texttt{minus}} & = & \texttt{function minus(x : nat, y : nat) : nat} \Leftarrow \\
& & \quad \texttt{if(?0(y),} \\
& & \qquad \texttt{x,} \\
& & \qquad \texttt{if(?0(x),} \\
& & \qquad\quad \texttt{0,} \\
& & \qquad\quad \texttt{minus(pred(x), pred(y))))}
\end{array}
$$

$$
\begin{array}{rcl}
D_{\texttt{quot}} & = & \texttt{function quot(x : nat, y : nat) : nat} \Leftarrow \\
& & \quad \texttt{if(?0(y),} \\
& & \qquad \texttt{x,} \\
& & \qquad \texttt{if(y>x,} \\
& & \qquad\quad \texttt{0,} \\
& & \qquad\quad \texttt{if?0(x),} \\
& & \qquad\qquad \texttt{0,} \\
& & \qquad\qquad \texttt{quot(minus(x, y), y))))}
\end{array}
$$

$$
\begin{array}{rcl}
D_{\texttt{rem}} & = & \texttt{function rem(x : nat, y : nat) : nat} \Leftarrow \\
& & \quad \texttt{if(?0(x),} \\
& & \qquad \texttt{0,} \\
& & \qquad \texttt{if(?0(y),} \\
& & \qquad\quad \texttt{0,} \\
& & \qquad\quad \texttt{if(y>x,} \\
& & \qquad\qquad \texttt{x,} \\
& & \qquad\qquad \texttt{rem(minus(x, y), y))))}
\end{array}
$$

Abbildung 9.1: Prozedurdefinitionen für `minus`, `quot` und `rem`.

Zum Beweis dieses Lemmas ist es notwendig, die Gültigkeit der Sequenz

$$\emptyset, \emptyset \vdash \texttt{if(?0(y), true, if(?0(pred(y)), ?0(rem(x, y)), true))}$$

nachzuweisen. Der Teilterm `rem(x, y)` schlägt hierzu eine Induktion über die Relationenbeschreibung R vor. Für diese Induktion erhalten wir als Induktionsschritt die Sequenz

$$H, IH \vdash \texttt{if(?0(y), true, if(?0(pred(y)), ?0(rem(x, y)), true))} \tag{9.7}$$

mit

$$
\begin{array}{rcl}
H & = & \{\neg\texttt{?0(x)}, \neg\texttt{?0(y)}\} \\
IH & = & \{\texttt{all y}' : \texttt{nat} \\
& & \quad \texttt{if(?0(y}'\texttt{), true, if(?0(pred(y}'\texttt{)), ?0(rem(minus(x, y}'\texttt{), y}'\texttt{)), true))}\}\,.
\end{array}
$$

Es ergibt sich dann die in Abbildung 9.2 dargestellte symbolische Auswertung. Hierbei haben wir zur Auswertung von Prozeduraufrufen die Heuristik aus Abschnitt 9.2.1 verwendet. Weiter haben wir angenommen, dass wir den Prozeduraufruf `pred(x)>pred(y)` auswerten dürfen. Der Term

$$\texttt{if(?0(pred(y)), ?0(rem(minus(pred(x), pred(y)), y)), true)} \tag{9.8}$$

ist nicht weiter auswertbar und stellt somit das Ergebnis der symbolischen Auswertung dar. Für den Term (9.8) ist offensichtlich die Induktionshypothese nicht anwendbar. Wir haben somit unser Ziel, den Goal-Term durch symbolische Auswertung auf die Induktionshypothese zu reduzieren, nicht erreicht. Das Problem

$\mathcal{U} \vdash$ `if(?0(y), true, if(?0(pred(y)), ?0(rem(x, y)), true))`

$\xrightarrow{(1)}$ `if(false, true, if(?0(pred(y)), ?0(rem(x, y)), true))`

$\xrightarrow{(2)}$ `if(?0(pred(y)), ?0(rem(x, y)), true)`

$\xrightarrow{(3)}$ `if(?0(pred(y)), ?0(if(?0(x), 0, if(?0(y), 0, if(y>x, x, rem(minus(x, y), y))))), true)`

$\xrightarrow{(4)}$ `if(?0(pred(y)), ?0(if(false, 0, if(?0(y), 0, if(y>x, x, rem(minus(x, y), y))))), true)`

$\xrightarrow{(5)}$ `if(?0(pred(y)), ?0(if(?0(y), 0, if(y>x, x, rem(minus(x, y), y)))), true)`

$\xrightarrow{(6)}$ `if(?0(pred(y)), ?0(if(false, 0, if(y>x, x, rem(minus(x, y), y)))), true)`

$\xrightarrow{(7)}$ `if(?0(pred(y)), ?0(if(y>x, x, rem(minus(x, y), y))), true)`

$\xrightarrow{(8)}$ `if(?0(pred(y)), ?0(if(if(?0(y), false, if(?0(x), true, pred(y)>pred(x))), x, rem(minus(x, y), y))), true)`

$\xrightarrow{(9)}$ `if(?0(pred(y)), ?0(if(if(false, false, if(?0(x), true, pred(y)>pred(x))), x, rem(minus(x, y), y))), true)`

$\xrightarrow{(10)}$ `if(?0(pred(y)), ?0(if(if(?0(x), true, pred(y)>pred(x)), x, rem(minus(x, y), y))), true)`

$\xrightarrow{(11)}$ `if(?0(pred(y)), ?0(if(if(false, true, pred(y)>pred(x)), x, rem(minus(x, y), y))), true)`

$\xrightarrow{(12)}$ `if(?0(pred(y)), ?0(if(pred(y)>pred(x), x, rem(minus(x, y), y))), true)`

$\xrightarrow{(13)}$ `if(?0(pred(y)), ?0(if(if(?0(pred(y)), false, if(?0(pred(x)), true, pred(pred(y))>pred(pred(x)))), x, rem(minus(x, y), y))), true)`

$\xrightarrow{(14)}$ `if(?0(pred(y)), ?0(if(if(true, false, if(?0(pred(x)), true, pred(pred(y))>pred(pred(x)))), x, rem(minus(x, y), y))), true)`

$\xrightarrow{(15)}$ `if(?0(pred(y)), ?0(if(if(false, x, rem(minus(x, y), y)), x, rem(minus(x, y), y))), true)`

$\xrightarrow{(16)}$ `if(?0(pred(y)), ?0(rem(minus(x, y), y)), true)`

$\xrightarrow{(17)}$ `if(?0(pred(y)), ?0(rem(if(?0(x), 0, if(?0(y), x, minus(pred(x), pred(y)))), y)), true)`

$\xrightarrow{(18)}$ `if(?0(pred(y)), ?0(rem(if(false, 0, if(?0(y), x, minus(pred(x), pred(y)))), y)), true)`

$\xrightarrow{(19)}$ `if(?0(pred(y)), ?0(rem(if(?0(y), x, minus(pred(x), pred(y))), y)), true)`

$\xrightarrow{(20)}$ `if(?0(pred(y)), ?0(rem(if(false, x, minus(pred(x), pred(y))), y)), true)`

$\xrightarrow{(21)}$ `if(?0(pred(y)), ?0(rem(minus(pred(x), pred(y)), y)), true)`

Abbildung 9.2: Symbolische Auswertung des Goal-Terms der Sequenz (9.7) aus Beispiel 9.8. Es wird angenommen, dass der Prozeduraufruf `pred(x)>pred(y)` in Auswertungsschritt (13) ausgewertet werden darf.

ist, dass wir in Auswertungsschritt (17) das Argument $\texttt{minus}(\texttt{x},\texttt{y})$ des rekursiven Aufrufs von `rem` auswerten. Ohne diese Auswertung würde die symbolische Auswertung den Term

$$\texttt{if}(\texttt{?0}(\texttt{pred}(\texttt{y})),\texttt{?0}(\texttt{rem}(\texttt{minus}(\texttt{x},\texttt{y}),\texttt{y})),\texttt{true})$$

als Ergebnis liefern und wir könnten die Induktionshypothese anwenden. Wir bezeichnen die symbolische Auswertung aus Abbildung 9.2 daher auch als *Überauswertung.* □

Das Beispiel zeigt, dass es nicht sinnvoll ist Argumente von rekursiven Aufrufen auszuwerten. Solche Auswertungen können nicht nur, wie in Beispiel 9.8, zu Termen führen, auf die die Induktionshypothese nicht mehr anwendbar ist, sondern auch zu Endlos-Auswertungen (siehe Beispiel F.6 in Anhang F). Um solche Auswertungen zu vermeiden, müssen wir die Auswertung von Prozeduraufrufen in den Argumenten der rekursiven Aufrufe verhindern. Dies betrifft jedoch nur Prozeduraufrufe in den Argumenten von Induktionsvariablen der Prozedur f, da nur die Auswertung dieser Prozeduraufrufe zu Überauswertungen und Endlos-Auswertungen führen kann. Wir formulieren daher die folgende Anforderung:

Anforderung (Auswertung rekursiver Argumente):
Bei der Auswertung eines instantiierten Prozedurrumpfs $R_f^*[x^*/t^*]$ einer rekursiv-definierten Prozedur f dürfen die Prozeduraufrufe in den Argumentpositionen von Induktionsvariablen in den rekursiven Aufrufen von f nicht ausgewertet werden.

Zur Erfüllung dieser Anforderung beschränken wir zunächst allgemein die Auswertung von Prozeduraufrufen auf Basis unserer Heuristik. Wir werten einen Prozeduraufruf

$$g^{\langle\ldots,l,\ldots\rangle}(t_1,\ldots,t_n)$$

nur noch dann auf Basis unserer Heuristik aus, falls das Label l nicht in der aktuellen Kontextmenge C der A-Umgebung enthalten ist. Weiter annotieren wir den Rumpf R_f^* einer Prozedur f wie folgt: Für jeden rekursiven Aufruf $f(r_1,\ldots,r_n)$ in R_f^* setzen wir das Label eines Prozeduraufrufs $g(s^*)$ in den Argumenten r_i der Induktionsvariablen von f auf die Prozedur h, falls der Prozeduraufruf innerhalb eines Prozeduraufrufs von h in r_i auftaucht. Das bedeutet, das Argument r_i hat folgende Form:

$$\ldots h(\ldots g(s^*)\ldots)\ldots$$

Für alle anderen Prozeduraufrufe setzen wir das Label auf $\bot$. Wir sperren dadurch die Auswertung der Prozeduraufrufe $g(s^*)$ in den Argumenten r_i solange, wie der Prozeduraufruf innerhalb eines Prozeduraufrufs von h enthalten ist. Für den Prozedurrumpf von `rem` aus Beispiel 9.8 ergibt sich dann folgender annotierter Rumpf:[4]

$$\begin{array}{l}
\texttt{if}^{\bot}(\texttt{?0}^{\bot}(\texttt{x}),\\
\quad \texttt{0}^{\bot},\\
\quad \texttt{if}^{\bot}(\texttt{?0}^{\bot}(\texttt{y}),\\
\qquad \texttt{0}^{\bot},\\
\qquad \texttt{if}^{\bot}(\texttt{y}>^{\bot}\texttt{x},\\
\qquad\quad \texttt{x},\\
\qquad\quad \texttt{rem}^{\bot}(\texttt{minus}^{\texttt{rem}}(\texttt{x},\texttt{y}),\texttt{y})))).
\end{array}$$

[4] Wir stellen der Übersichtlichkeit halber lediglich die Labels der A-Annotationen dar.

Weiter erhalten wir so für den Prozedurrumpf von `mult` aus Abbildung 8.2 den annotierte Rumpf

$$\begin{array}{l}\texttt{if}^{\,\bot}(\texttt{?0}^{\,\bot}(\texttt{x}),\\ \quad\texttt{if}^{\,\bot}(\texttt{even}^{\,\bot}(\texttt{x}),\\ \quad\quad\texttt{mult}^{\,\bot}(\texttt{half}^{\,\texttt{mult}}(\texttt{x}),\texttt{dbl}^{\,\bot}(\texttt{y})),\\ \quad\quad\texttt{plus}^{\,\bot}(\texttt{mult}^{\,\bot}(\texttt{half}^{\,\texttt{mult}}(\texttt{x}),\texttt{dbl}^{\,\bot}(\texttt{y})),\texttt{y})))\end{array}$$

und für den Prozedurrumpf von `minsort` aus Abbildung 9.3 den annotierten Rumpf

$$\begin{array}{l}\texttt{if}^{\,\bot}(\texttt{?empty}^{\,\bot}(\texttt{l}),\\ \quad\texttt{empty}^{\,\bot},\\ \quad\texttt{add}^{\,\bot}(\texttt{minimum}^{\,\bot}(\texttt{l}),\\ \quad\quad\texttt{minsort}^{\,\bot}(\texttt{delete}^{\,\texttt{minsort}}(\texttt{minimum}^{\,\texttt{delete}}(\texttt{l}),\texttt{l})))).\end{array} \tag{9.9}$$

Um einen Prozedurrumpf R_f^* einer Prozedur f auf diese Weise zu annotieren definieren wir die Funktion

$$\mathbf{u}_f^{\delta,g} : \mathcal{T}erm(P) \to \mathcal{AT}erm(P).$$

Hierbei bezeichnet g ein Element der Menge $\Sigma^{\mathrm{proc}}(P) \cup \{\bot\}$ und δ eine Termsubstitution. Um die Definition der Funktion $\mathbf{u}_f^{\delta,g}$ übersichtlich zu halten, gehen wir davon aus, dass die Prozedur f durch eine Prozedurdefinition der Form

$$\texttt{function}\ f(x^* : \tau^*, y^* : \overline{\tau}^*) : \tau \ \Leftarrow\ R_f$$

eingeführt wurde. Weiter gehen wir davon aus, dass die Parameter x^* die Induktionsvariablen der Prozedur bezeichnen, d.h. es gilt

$$\{x^*\} = \bigcup_{R \in grds(f)} iv(R).$$

Das Funktionssymbol h bezeichne in der nachfolgenden Definition immer ein Funktionssymbol ungleich f. Die Funktion $\mathbf{u}_f^{\delta,g}$ ist dann wie folgt definiert:

$$\begin{array}{rcll}
\mathbf{u}_f^{\delta,g}(x) & := & x & \text{falls } g = \bot \text{ oder } x \notin dom(\delta),\\
\mathbf{u}_f^{\delta,g}(x) & := & \mathbf{u}_f^{\delta,g}(\delta(x)) & \text{falls } g \neq \bot \text{ und } x \in dom(\delta),\\
\mathbf{u}_f^{\delta,g}(h(t^*)) & := & h^{\,A_{\mathrm{std}}}(\mathbf{u}_f^{\delta,g}(t^*)) & \text{falls } h \notin \Sigma^{\mathrm{proc}}(P),\\
\mathbf{u}_f^{\delta,g}(h(t^*)) & := & h^{\,A^g_{\mathrm{std}}}(\mathbf{u}_f^{\delta,g}(t^*)) & \text{falls } h \in \Sigma^{\mathrm{proc}}(P) \text{ und } g = \bot,\\
\mathbf{u}_f^{\delta,g}(h(t^*)) & := & h^{\,A^g_{\mathrm{std}}}(\mathbf{u}_f^{\delta,h}(t^*)) & \text{falls } h \in \Sigma^{\mathrm{proc}}(P) \text{ und } g \neq \bot,\\
\mathbf{u}_f^{\delta,g}(f(t^*,r^*)) & := & f^{\,A^g_{\mathrm{std}}}(\mathbf{u}_f^{\delta,f}(t^*),\mathbf{u}_f^{\delta,g}(r^*)), & \\
\mathbf{u}_f^{\delta,g}(\texttt{let}\ x\texttt{:=}t\ \texttt{in}\ r\ \texttt{end}) & := & \texttt{let}^{\,A_{\mathrm{std}}}\ x\texttt{:=}\mathbf{u}_f^{\delta,g}(t)\ \texttt{in}\ \mathbf{u}_f^{\delta\oplus\{x/\delta(t)\},g}(r)\ \texttt{end}. &
\end{array}$$

Hierbei bezeichnet A^g_{std} die folgende A-Annotation:

	Pol.	Mark.	Lab.	U-Lim.	S-Lim.	Inst.
A^g_{std}:	$\odot$	0	g	0	0	$\bot$

Wir ersetzen dann bei der Auswertung eines Prozeduraufrufs $f(t_1, \ldots, t_n)$ diesen Prozeduraufruf durch den folgenden annotierten und instantiierten Prozedurrumpf:

$$\mathbf{u}_f^{\epsilon,\perp}(R_f^*)[x_1/t_1, \ldots, x_n/t_n].$$

Die Funktion $\mathbf{u}_f^{\delta,g}$ behandelt `let`-Terme auf spezielle Weise. Zur Erklärung betrachten wir die Prozedurdefinitionen der Prozeduren `minsort` und `minsort'` aus Abbildung 9.3. Der durch $\mathbf{u}_{\texttt{minsort}}^{\epsilon,\perp}$ annotierte Prozedurrumpf von `minsort` ist durch den Term (9.9) gegeben. Die beiden Prozeduraufrufe von `minimum(l)` in (9.9) wurden mit verschiedenen Labels annotiert: Der Prozeduraufruf außerhalb des rekursiven Aufrufs wurde mit $\perp$ annotiert und der Prozeduraufruf innerhalb des rekursiven Aufrufs mit `minsort`. Eine Auswertung des Prozeduraufrufs `minimum(l)` außerhalb des rekursiven Aufrufs ist also möglich. Betrachten wir nun die Prozedurdefinition von `minsort'`. Die Prozedurrümpfe der Prozeduren `minsort` und `minsort'` unterscheiden sich lediglich dadurch, das für den Term `minimum(l)` im Prozedurrumpf von `minsort'` die Variablenbindung `m=minimum(l)` eingeführt wurde. Um zu gewährleisten, dass die Prozedurrümpfe von `minsort` und `minsort'` auf ähnliche Weise annotiert und damit in ähnlicher Weise ausgewertet werden, setzen wir durch die Funktion $\mathbf{u}_{\texttt{minsort}'}^{\delta,g}$ die Variablenbindung `m=minimum(l)` in das Argumente der Induktionsvariablen des rekursiven Aufrufs von `minsort'` ein. Zum Einsetzen der Variablenbindung verwenden wir die Substitution δ. Es ergibt sich dann folgender annotierter Prozedurrumpf für die Prozedur `minsort'`:

```
if^⊥(?empty^⊥(l),
     empty^⊥,
     let m:=minimum^⊥(l) in
         add^⊥(m, minsort'^⊥(delete^minsort'(minimum^delete(l), l)))
     end).
```

Dadurch, dass wir die Variablenbindungen `m=minimum(l)` in den rekursiven Aufruf von `minsort'` einsetzen, ist es möglich, die Auswertung von `minimum(l)` außerhalb des rekursiven Aufrufs zu erlauben und lediglich die Auswertung innerhalb des rekursiven Aufrufs zu sperren.[5]

9.2.3 Weitere Annotationen des Prozedurrumpfs

Durch die Funktion $\mathbf{u}_f^{\epsilon,\perp}$ werden nur die Labels der Prozeduraufrufe im Prozedurrumpf korrekt gesetzt. Für eine effiziente symbolische Auswertung ist es jedoch wichtig, dass sich das Search-Limit und das Unfold-Limit des Prozeduraufrufs auf die Funktionsaufrufe des Prozedurrumpfs vererbt. Wir definieren daher zur vollständigen Annotierung des Prozedurrumpfs die folgende Funktion:[6]

$$\mathbf{un}_f^{s,u,m}(t) \quad := \quad \mathbf{a}_A(\mathbf{u}_f^{\epsilon,\perp}(t)) \quad \text{mit}$$

	Pol.	Mark.	Lab.	U-Lim.	S-Lim.	Inst.
A:	$-$	m	$-$	u	s	$-$

[5] `let`-Ausdrücke werden verwendet, um Mehrfachauswertungen von Ausdrücken zu vermeiden. Folglich ist das Einsetzen von `let`-Bindungen unter diesem Aspekt unsinnig. Hier geht es jedoch nicht um Berechnungen, sondern um die Verwaltung von Kontrollinformationen für die automatische Beweisführung. In diesem Zusammenhang ist - wie das Beispiel illustriert - die Auflösung von `let`-Bindungen durchaus sinnvoll, um Terme geeignet mit Kontrollinformationen zu versehen.

[6] Die Bedeutung der Markierung m erklären wir in Abschnitt 10.

$D_{\texttt{delete}}$ =
```
function delete(n : @a, l : list[@a]) : list[@a] ⇐
  if(?empty(l),
     empty,
     if(n=hd(l),
        tl(k),
        add(n, delete(n, tl(k)))))
```

$D_{\texttt{minimum}}$ =
```
function minimum(l : list[nat]) : nat ⇐
  if(?empty(l),
     *,
     if(?empty(tl(l)),
        hd(l),
        if(hd(l)>minimum(tl(l)),
           minimum(tl(l)),
           hd(l))))
```

$D_{\texttt{ordered}}$ =
```
function ordered(l : list[nat]) : bool ⇐
  if(?empty(l),
     true,
     if(?empty(tl(l)),
        true,
        if(hd(l)>hd(tl(l)),
           false,
           ordered(tl(k)))))
```

$D_{\texttt{minsort}}$ =
```
function minsort(l : list[nat])list[nat] ⇐
  if(?empty(l),
     empty,
     add(minimum(l), minsort(delete(minimum(l), l))))
```

$D_{\texttt{minsort}'}$ =
```
function minsort'(l : list[nat])list[nat] ⇐
if(?empty(l),
   empty,
   let m:=minimum(l) in
       add(m, minsort'(delete(m, l)))
    end)
```

Abbildung 9.3: Prozedurdefinitionen für `delete`, `minimum`, `ordered`, `minsort` und `minsort'`.

$\mathcal{U} \vdash$ `if(?0(n), true, log2(dbl(n))=succ(log2(n)))` $\overset{(1)}{\rightarrow}^*$

`log2(dbl(n))=succ(log2(n))` $\overset{(2)}{\rightarrow}^*$

```
if(?0(pred(n)),
   true,
   log2(succ(succ(dbl(pred(n)))))=succ(log2(n)))
```
$\overset{(3)}{\rightarrow}^*$

```
if(?0(pred(n)),
   true,
   log2(half(succ(succ(dbl(pred(n))))))=succ(log2(n)))
```
$\overset{(4)}{\rightarrow}^*$

```
if(?0(pred(n)),
   true,
   if(?0(pred(half(succ(succ(dbl(pred(x))))))),
      false,
      log2(half(half(succ(succ(dbl(pred(x))))))=log2(half(x))))
```

Abbildung 9.4: Symbolische Auswertung des Goal-Terms der Sequenz (9.10).

Wie wir bereits zu Beginn des Kapitels erwähnt haben, wird ein Prozeduraufruf $f(t_1, \ldots, t_n)$ erst dann ausgewertet, wenn alle Argumente $t_1, \ldots, t_n$ des Prozeduraufrufs ausgewertet sind. Wir können daher, ohne die Auswertung des instantiierten Prozedurrumpfs

$$R_f^*[x_1/t_1, \ldots, x_n/t_n]$$

spürbar zu beschränken, das Search-Limit und das Unfold-Limit in den Argumenten $t_1, \ldots, t_n$ für die Instantiierung des Prozedurrumpfs R_f^* auf 0 setzen. Dadurch werden für die Argumente $t_1, \ldots, t_n$ keine Quantifikationen und keine „*Functionality*"-Regeln angewendet. Weiter wird die Anwendung der „*Unfold*"-Regeln verhindert. Diese Einschränkungen sind für die Effizienz des A-Kalküls ausgesprochen wichtig.

9.2.4 Auswertung der Recursion-Guards

Die Auswertung von Prozeduraufrufen in den Argumenten $t_1, \ldots, t_n$ eines Prozeduraufrufs $f(t_1, \ldots, t_n)$ während der Auswertung des instantiierten Recursion-Guards kann zu Überauswertungen führen. Betrachten wir dazu ein Beispiel:

Beispiel 9.9. Sei das Programm P gegeben durch die Prozedurdefinitionen $D_{\texttt{dbl}}$, $D_{\texttt{half}}$ aus Abbildung 2.11 und der folgenden Prozedurdefinition

```
function log2(n : nat) : nat ⇐
  if(?0(n),
     0,
     if(?0(pred(x)),
        0,
        succ(log2(half(x))))).
```

Der Recursion-Guard von `log2` sei gegeben durch $\neg$`?(n)`. Wir wollen nun das folgende Lemma durch Induktion über `log2` beweisen:

```
lemma log2_dbl2 ⇐ all n : nat
  if(?0(n), true, log2(dbl(n))=succ(log2(n))).
```

Für den Induktionsschritt erhalten dabei die folgende Sequenz:

$$\begin{array}{c}\{\neg\texttt{?0(n)}\},\\ \{\texttt{if(?0(half(n)),true,log2(dbl(half(n)))=succ(log2(half(n))))}\} \vdash \\ \texttt{if(?0(n),true,log2(dbl(n))=succ(log2(n)))}.\end{array} \tag{9.10}$$

Für den Goal-Term dieser Sequenz ergibt sich die in Abbildung 9.4 dargestellte symbolische Auswertung. Es ist zu beachten, dass der Prozeduraufruf in Auswertungsschritt (4) ausgewertet werden darf, da der instantiierte Recursion-Guard

$$\neg\texttt{?(half(succ(succ(dbl(pred(n))))))} \tag{9.11}$$

zu `true` ausgewertet werden kann. Diese Auswertung des Prozeduraufrufs in Auswertungsschritt (4) führt zur Überauswertung des Goal-Terms. Das bedeutet, dass der die Induktionshypothese nicht auf das Ergebnis der symbolischen Auswertung angewendet werden kann. Die Auswertung des instantiierten Recursion-Guard (9.11) zu `true` ist möglich, da der Prozeduraufruf `half(succ(succ(dbl(pred(n)))))` in (9.11) ausgewertet werden kann. □

Um Überauswertungen wie in Beispiel 9.9 zu verhindern, sperren wir die Auswertung von Prozeduraufrufen in den Argumenten $t_1, \ldots, t_n$ während der Auswertung des Recursion-Guards. Wir annotieren dazu die Prozeduraufrufe in den Argumenten $t_1, \ldots, t_n$ im instantiierten Recursion-Guard $guard[x_1/t_1, \ldots, x_n/t_n]$ mit dem Label $\top$.

Überauswertungen sind auch dann möglich, wenn Quantifikationen zur Auswertung des instantiierten Recursion-Guards verwendet werden. Ist beispielsweise während der symbolischen Auswertung der Sequenz (9.10) aus Beispiel 9.9 die Gültigkeit des Lemmas

$$\begin{array}{l}\texttt{lemma log2\_stupid} \Leftarrow \texttt{all n : nat}\\ \quad\texttt{if(?0(n),true,}\neg\texttt{?(half(succ(succ(dbl(pred(n))))))}\end{array} \tag{9.12}$$

bekannt und annotieren wir, wie erwähnt, die Prozeduraufrufe in den Argumenten $t_1, \ldots, t_n$ im instantiierten Recursion-Guard $guard[x_1/t_1, \ldots, x_n/t_n]$ mit dem Label $\top$, so kann der instantiierte Recursion-Guard (9.11) mit Hilfe des Lemmas (9.12) zu `true` vereinfacht werden. Es ergibt sich dann wieder die Überauswertung aus Abbildung 9.4. Wir setzen daher zur Auswertung des instantiierten Recursion-Guards zusätzlich das Search-Limit jedes Teilterms auf 0 und sperren damit die Verwendung von Quantifikationen.

9.2.5 Unvollständig-definierte Prozeduren

Für unvollständig definierte Prozeduren f müssen wir im Prozedurrumpf R_f^* die rekursiven Aufrufe für eine symbolische Auswertung sperren, die durch das Symbol $*$ in den Prozedurrumpf R_f^* eingeführt wurden, da ansonsten Endlos-Auswertungen entstehen.[7] Hierzu definieren wir die folgende Funktion:

$$\begin{array}{rcl}\mathtt{s}_f^*(x) & := & x,\\ \mathtt{s}_f^*(*^{\,A}) & := & f^{\,A_{\text{top}}}(x_1, \ldots, x_n),\\ \mathtt{s}_f^*(g^{\,A}(t_1, \ldots, t_n)) & := & g^{\,A}(\mathtt{s}_f^*(t_1), \ldots, \mathtt{s}_f^*(t_n))),\\ \mathtt{s}_f^*(\texttt{let}^{\,A}\ x\texttt{:=}t\ \texttt{in}\ r\ \texttt{end}) & := & \texttt{let}^{\,A}\ x\texttt{:=}\mathtt{s}_f^*(t)\ \texttt{in}\ \mathtt{s}_f^*(r)\ \texttt{end}.\end{array}$$

[7] Die Verwendung des Exception-Guard ist für die Regel „*Execute procedure call (recursive cases)*“ ungeeignet, da der Exception-Guard in den meisten Fällen nicht zu `false` ausgewertet werden kann und wir somit die Regel praktisch für unvollständig-definierte Prozeduren niemals anwenden würden.

Wir interpretieren dann einen Prozeduraufruf $f^{\,A}(t_1, \ldots, t_n)$ durch den Term $\sigma(\mathsf{s}^*_f(\mathsf{un}^{u,s,0}_f(R_f)))$.

9.2.6 Vollständige Regeldefinition

Insgesamt ergibt sich aufgrund der vorhergehenden Abschnitte die folgende Definition der Regel „*Execute procedure call (recursive cases)*“:

43. Execute procedure call (recursive cases)

$$\frac{f^{\,A}(t_1, \ldots, t_n)}{\sigma_{\xi_{f(\ldots)}}(\mathsf{s}^*_f(\mathsf{un}^{u,s,0}_f(R_f)))}$$

	Label	S-Lim.	U-Lim.
A:	l	s	u

falls $f \in \Sigma \cap \Sigma^{\mathrm{rec}}(P)$, $l \notin C$, $\sigma = \{x_1/\mathsf{a}_{A_0}(t_1), \ldots, x_n/\mathsf{a}_{A_0}(t_n)\}$ und eine *guard* $\in$ *rec-guards*(f) existiert mit

$$\mathcal{U}' \vdash \mathsf{a}_{A_{\mathrm{std}}}(\mathit{guard})[x_1/\mathsf{a}_{A_{\mathrm{top}}}(t_1), \ldots, x_n/\mathsf{a}_{A_{\mathrm{top}}}(t_n)] \to^{!} \mathtt{true}.$$

Hierbei bezeichnet $\mathcal{U}'$ die folgende A-Umgebung:

	$\mathrm{Hyp}_{\mathrm{L}}$	Termsig.
$\mathcal{U}'$:	$\emptyset$	$\Sigma|_{dom(\Sigma)\setminus\{f\}}$

9.3 Auswertung zu Basisfällen

Die Regel „*Execute procedure call (recursive cases)*“ wurde zur Auswertung von Prozeduraufrufen definiert, um den Goal-Term an die Induktionshypothesen anzunähern. Hierzu wertet die Regel einen Prozeduraufruf aus, falls gewährleistet ist, dass sich der Prozedurrumpf zu entsprechenden rekursiven Aufrufen vereinfachen lässt. Insbesondere für eine erfolgreiche symbolische Auswertung des Goal-Terms eines Induktionsbasisfalls ist es jedoch erforderlich, Prozeduraufrufe auch dann auszuwerten, falls sich die Prozeduraufrufe auf Basisfälle der Prozeduren reduzieren lassen. Betrachten wir dazu ein Beispiel:

Beispiel 9.10. Betrachten wir erneut das Lemma `plus_neutral` aus Beispiel 8.9. Eine Induktion über die Relationenbeschreibung

$$R \;=\; \{\langle\{\neg\mathtt{?0(x)}\}, \{\{\mathtt{x/pred(x)}\}\}\rangle\}$$

liefert die folgende Sequenz als Induktionsbasisfall:

$$\{\mathtt{?0(x)}\}, \emptyset \vdash \mathtt{plus(x, 0)}.$$

Durch Auswertung des Prozeduraufrufs `plus(x, 0)` können wir `plus(x, 0)` auf den Basisfall der Prozedur `plus` reduzieren und so die Gültigkeit des Basisfalls beweisen:

$$
\begin{array}{lll}
\mathcal{U} \vdash & \underline{\texttt{plus(x,0)=x}} & \rightarrow \\
& \texttt{if(}\underline{\texttt{?0(x)}}\texttt{,succ(plus(pred(x),0)),0)=x} & \rightarrow \\
& \underline{\texttt{if(true,succ(plus(pred(x),0)),0)=x}} & \rightarrow \\
& \underline{\texttt{0=x}} & \rightarrow \\
& \texttt{true}. &
\end{array}
$$

□

Um solche Auswertungen zu ermöglichen, definieren wir in diesem Abschnitt die Regel „*Execute procedure call (non recursive cases)*". Diese Regel ist auf Basis der folgenden Heuristik definiert:

Heuristik (Auswertung zu Basisfällen):
Kann ein Prozeduraufruf $f(t^*)$ einer rekursiv-definierten Prozedur f auf Basisfälle von f reduziert werden, so wird der Prozeduraufruf ausgewertet.

Um auf effiziente Weise zu überprüfen, ob ein Prozeduraufruf $f(t^*)$ auf die Basisfälle der Prozedur f reduziert werden kann, definieren wir für jede rekursiv-definierte Prozedur g den so genannten *Execution-Guard*. Der Execution-Guard einer Prozedur g definiert die Bedingungen, die die Argumente s^* eines Prozeduraufrufs $g(s^*)$ erfüllen müssen, damit der Prozeduraufruf auf die Basisfälle der Prozedur g reduziert werden kann. Um den Execution-Guard einer Prozedur definieren zu können, müssen wir zunächst formal festlegen, was wir unter einem Basisfall verstehen wollen. Wir führen hierzu die Begriffe der *Ergebnisposition* und des *Ergebnisterms* einer Prozedurdefinition ein:

Definition 9.11 (Ergebnispositionen und Ergebnisterme)**.** Sei D eine Prozedurdefinition der Form

$$\texttt{function}\ f(x^* : \tau^*) : \tau \ \Leftarrow\ R_f.$$

Eine Position $\pi \in \mathcal{Pos}(|R_f|_{\mathcal{P}})$ heißt *Ergebnisposition von* D falls $|R_f|_{\mathcal{P}}\big|_\pi$ `case`-frei ist und die folgende Implikation gilt:

$$\pi = \pi' i \implies |R_f|_{\mathcal{P}}\big|_{\pi'} = \texttt{case}(\ldots) \text{ und } i > 1.$$

Die Menge aller Ergebnispositionen von D wird mit $\mathcal{Pos}_{\mathrm{E}}(D)$ bezeichnet. Für eine Ergebnisposition π von D wird der Term $|R_f|_{\mathcal{P}}\big|_\pi$ *Ergebnisterm von* D genannt. □

Ergebnispositionen bzw. Ergebnisterme repräsentieren genau die Terme, die die Resultate der Prozedur berechnen. Ist für eine Prozedur f die Prozedurdefinition D aus dem Zusammenhang klar, so sprechen wir auch von Ergebnisposition und Ergebnisterm der Prozedur f. Wir schreiben dann statt $\mathcal{Pos}_{\mathrm{E}}(D)$ auch einfach $\mathcal{Pos}_{\mathrm{E}}(f)$.

Beispiel 9.12. Die Ergebnispositionen der Prozedurdefinition von `minimum` aus Abbildung 9.3 lauten:

$$\langle 2\rangle, \langle 3,2\rangle, \langle 3,3,2\rangle, \langle 3,3,3\rangle.$$

□

Mit Hilfe von Ergebnispositionen und Ergebnistermen können wir nun die Positionen und Terme formal definieren, die die Basisfälle einer Prozedurdefinition repräsentieren. Diese Positionen und Terme bezeichnen wir als *Basispositionen* und *Basisterme* der Prozedurdefinition.

Definition 9.13 (Basispositionen und Basisterme)**.** Sei D eine Prozedurdefinition der Form

$$\texttt{function}\ f(x^* : \tau^*) : \tau \ \Leftarrow\ R_f\ .$$

Eine Ergebnisposition π von D heißt *Basisposition von* D, falls die Terme $|R_f|_{\mathcal{P}}\big|_\pi$ und $\mathit{cond}(\pi, |R_f|_{\mathcal{P}})$ keinen Prozeduraufruf von f enthalten. Die Menge aller Basispositionen von D wird mit $\mathcal{P}os_{\mathrm{B}}(D)$ bezeichnet. Für eine Basisposition π von D wird der Term $|R_f|_{\mathcal{P}}\big|_\pi$ *Basisterm von* D genannt. □

Basisterme sind genau die Ergebnisterme der Prozeduren, für die kein rekursiver Prozeduraufruf erforderlich ist, weder auf dem Pfad zum Ergebnisterm, noch im Ergebnisterm selbst. Ist für eine Prozedur f die Prozedurdefinition D aus dem Zusammenhang klar, so sprechen wir auch von Basisposition und Basisterm der Prozedur f. Wir schreiben dann statt $\mathcal{P}os_{\mathrm{B}}(D)$ auch einfach $\mathcal{P}os_{\mathrm{B}}(f)$. Es ist zu beachten, dass Ergebnisterme der Form $*$ einer Prozedur f als Basisterme erachtet werden. Dies ist korrekt, da der definierte, aber unbekannte Wert von $*$ immer ohne Rekursion berechnet werden kann.

Beispiel 9.14. Die Basispositionen der Prozedurdefinition von `minimum` aus Abbildung 9.3 lauten:

$$\langle 2\rangle, \langle 3, 2\rangle.$$

Es ist zu beachten, dass $\langle 3, 3, 2\rangle$ keine Basisposition ist, da die Bedingung

$$\texttt{hd(l)>minimum(tl(l))}$$

einen rekursiven Aufruf enthält. □

Nachdem nun geklärt ist, welche Positionen bzw. Terme als Basisfälle einer Prozedurdefinition zu betrachten sind, kommen wir zur Definition des Execution-Guards. Kann für einen Prozeduraufruf $\sigma(f(x_n, \ldots, x_n))$ und eine Basisposition π von f die Konjunktion $\sigma(\mathit{AND}(\mathit{cond}(\pi, |R_f|_{\mathcal{P}})))$ zu `true` ausgewertet werden, so lässt sich offensichtlich der instantiierte `let`-bereinigte Prozedurrumpf $\sigma(|R_f|_{\mathcal{P}})$ auf den entsprechenden Basisterm $\sigma(|R_f|_{\mathcal{P}})\big|_\pi$ reduzieren. Den nicht-`let`-bereinigten Prozedurrumpf $\sigma(R_f)$ können wir dann zu einem dem Basisterm entsprechenden Term vereinfachen. Die Konjunktion $\mathit{AND}(\mathit{cond}(\pi, |R_f|_{\mathcal{P}}))$ definiert damit die Bedingungen, die die Argumente von $\sigma(f(x_n, \ldots, x_n))$ erfüllen müssen, damit der Prozeduraufruf auf den Basisfall $|R_f|_{\mathcal{P}}\big|_\pi$ reduziert werden kann. Der Execution-Guard ist dann durch die Disjunktion all dieser Konjunktionen definiert:

Definition 9.15 (Execution-Guard)**.** Sei D eine rekursiv-definierte Prozedurdefinition der Form

$$\texttt{function}\ f(x^* : \tau^*) : \tau \ \Leftarrow\ R_f$$

und $\{\pi_1, \ldots, \pi_n\} = \mathcal{P}os_{\mathrm{B}}(D)$ mit $\pi_1 <_{\mathcal{P}os} \ldots <_{\mathcal{P}os} \pi_n$. Weiter sei

$$E = \mathit{OR}(\mathit{AND}(\mathit{cond}(\pi_1, |R_f|_{\mathcal{P}})), \ldots, \mathit{AND}(\mathit{cond}(\pi_n, |R_f|_{\mathcal{P}})))\,.$$

Der *Execution-Guard* exec_D *der Prozedurdefinition* D ist dann definiert als

$$\mathit{exec}_D := \mathsf{e}(\mathsf{a}_{A_{\mathrm{std}}}(E)\downarrow_{\mathcal{U}_0})\,.$$

□

Die symbolische Auswertung $\mathsf{a}_{A_{\mathrm{std}}}(E)\downarrow_{\mathcal{U}_0}$ der Disjunktion E in Definition 9.15 ist notwendig, um einen möglichst einfachen Execution-Guard zu erhalten. Insbesondere werden durch die symbolische Auswertung irrelevante Fallunterscheidungen aus E entfernt. Weiter ist es für eine effiziente Verwendung des Execution-Guards

erforderlich, dass die Konjunktionen $AND(cond(\pi_i, |R_f|_{\mathcal{P}}))$ in der Reihenfolge angeordnet sind, in der die zugehörigen Basisterme $|R_f|_{\mathcal{P}}\big|_{\pi_i}$ im Prozedurrumpf $|R_f|_{\mathcal{P}}$ auftauchen. Aus diesem Grund ordnen wir in Definition 9.15 die Basispositionen π_i mit Hilfe der Relation $>_{\mathcal{P}os}$ entsprechend an. Wir illustrieren die Berechnung des Execution-Guards anhand eines Beispiels.

Beispiel 9.16. Betrachten wir abermals die Prozedurdefinition von `minimum` aus Abbildung 9.3. Wie bereits erwähnt sind $\pi_1 = \langle 2 \rangle$ und $\pi_2 = \langle 3, 2 \rangle$ die beiden einzigen Basispositionen dieser Prozedurdefinition. Als Disjunktion der Terme $AND(cond(\pi_1, |R^*_{\texttt{minimum}}|_{\mathcal{P}}))$ und $AND(cond(\pi_2, |R^*_{\texttt{minimum}}|_{\mathcal{P}}))$ ergibt sich dann der Term

$$\texttt{if}(\texttt{?empty}(\texttt{l}), \texttt{true}, \texttt{if}(\neg\texttt{?empty}(\texttt{l}), \texttt{?empty}(\texttt{tl}(\texttt{l})), \texttt{false})).$$

Eine symbolische Auswertung dieses Terms liefert den Execution-Guard der Prozedur `minimum`:

$$exec_{\texttt{minimum}} = \texttt{if}(\texttt{?empty}(\texttt{l}), \texttt{true}, \texttt{?empty}(\texttt{tl}(\texttt{l}))).$$

□

Ein Prozeduraufruf $f(t^*)$ kann dann auf Basisfälle von f reduziert werden, falls der Term

$$exec_f[x^*/t^*] \tag{9.13}$$

zu `true` ausgewertet werden kann. Aus Effizienzgründen verwenden wir zur Auswertung des Terms (9.13) keine Quantifikationen und keine „*Functionality*“-Regeln. Weiter werten wir lediglich triviale Prozeduraufrufe aus. Mit diesen Begriffen können wir jetzt die Regel „*Execute procedure call (non recursive cases)*“ definieren:

45. Execute procedure call (non recursive cases)

$$\frac{f^{\,A}(t_1, \ldots, t_n)}{\sigma_{\xi_{f(\ldots)}}(\mathbf{s}^*_f(\mathbf{un}^{u,s,0}_f(R_f)))},$$

	Label	S-Lim.	U-Lim.
A:	l	s	u

falls $f \in \Sigma \cap \Sigma^{\mathrm{rec}}(P)$, $l \notin C$, $\sigma = \{x_1/\mathbf{a}_{A_0}(t_1), \ldots, x_n/\mathbf{a}_{A_0}(t_n)\}$ und es gilt

$$\mathcal{U} \vdash \mathbf{a}_{A_{\mathrm{top}}}(exec_f[x_1/\mathbf{e}(t_1), \ldots, x_n/\mathbf{e}(t_n)]) \to^! \texttt{true}\,.$$

Anmerkung 9.17. Die Annotationen des Terms $\sigma_{\xi_{f(\ldots)}}(\mathbf{s}^*_f(\mathbf{un}^{u,s}_f(R_f)))$ haben wir aus der Regel „*Execute procedure call (recursive cases)*“ übernommen. Die Annotationen sind aus denselben Gründen sinnvoll bzw. relevant wie für Regel „*Execute procedure call (recursive cases)*“. Es ist zu beachten, dass die Labels der Argumente der rekursiven Aufrufe irrelevant sind, da alle rekursiven Aufrufe aus dem Term $\sigma_{\xi_{f(\ldots)}}(\mathbf{s}^*_f(\mathbf{un}^{u,s}_f(R_f)))$ aufgrund der Anwendungsbedingung der Regel in nachfolgenden Vereinfachungen entfernt werden können. □

Betrachten wir abschließend noch ein Beispiel, um die Nützlichkeit der Regel zu demonstrieren.

Beispiel 9.18. Sei P das Programm $\langle D_{\texttt{minus}} \rangle$, wobei $D_{\texttt{minus}}$ die entsprechende Prozedurdefinition aus Abbildung 9.1 bezeichnet. Sei weiter die Menge $grds(\texttt{minus})$ der generalisierten Relationenbeschreibungen gegeben durch $\{R_1, R_2\}$, wobei gilt

$$\begin{aligned} R_1 &= \{\langle\{\neg\texttt{?0}(\texttt{x})\}, \{\{\texttt{x/pred}(\texttt{x})\}\}\rangle\}, \\ R_2 &= \{\langle\{\neg\texttt{?0}(\texttt{y})\}, \{\{\texttt{y/pred}(\texttt{y})\}\}\rangle\}. \end{aligned}$$

$\mathcal{U} \vdash$ $\mathtt{if}(\underline{\mathtt{x>y}}, \mathtt{pred}(\mathtt{minus}(\mathtt{x},\mathtt{y}))=\mathtt{minus}(\mathtt{pred}(\mathtt{x}),\mathtt{y}), \mathtt{true})$ $\rightarrow$

$\mathtt{if}(\mathtt{if}(\mathtt{?0}(\mathtt{x}), \mathtt{false}, \mathtt{if}(\underline{\mathtt{?0}(\mathtt{y})}, \mathtt{true}, \mathtt{pred}(\mathtt{x})>\mathtt{pred}(\mathtt{y}))), \mathtt{pred}(\mathtt{minus}(\mathtt{x},\mathtt{y}))=\mathtt{minus}(\mathtt{pred}(\mathtt{x}),\mathtt{y}), \mathtt{true})$ $\rightarrow$

$\mathtt{if}(\mathtt{if}(\mathtt{?0}(\mathtt{x}), \mathtt{false}, \underline{\mathtt{if}(\mathtt{true}, \mathtt{true}, \mathtt{pred}(\mathtt{x})>\mathtt{pred}(\mathtt{y}))}), \mathtt{pred}(\mathtt{minus}(\mathtt{x},\mathtt{y}))=\mathtt{minus}(\mathtt{pred}(\mathtt{x}),\mathtt{y}), \mathtt{true})$ $\rightarrow$

$\underline{\mathtt{if}(\mathtt{if}(\mathtt{?0}(\mathtt{x}), \mathtt{false}, \mathtt{true}), \mathtt{pred}(\mathtt{minus}(\mathtt{x},\mathtt{y}))=\mathtt{minus}(\mathtt{pred}(\mathtt{x}),\mathtt{y}), \mathtt{true})}$ $\rightarrow$

$\mathtt{if}(\mathtt{?0}(\mathtt{x}), \mathtt{true}, \mathtt{pred}(\underline{\mathtt{minus}(\mathtt{x},\mathtt{y})})=\mathtt{minus}(\mathtt{pred}(\mathtt{x}),\mathtt{y}))$ $\rightarrow$

$\mathtt{if}(\mathtt{?0}(\mathtt{x}), \mathtt{true}, \mathtt{pred}(\mathtt{if}(\underline{\mathtt{?0}(\mathtt{x})}, \mathtt{0}, \mathtt{if}(\mathtt{?0}(\mathtt{y}), \mathtt{x}, \mathtt{minus}(\mathtt{pred}(\mathtt{x}), \mathtt{pred}(\mathtt{y})))))=\mathtt{minus}(\mathtt{pred}(\mathtt{x}),\mathtt{y}))$ $\rightarrow$

$\mathtt{if}(\mathtt{?0}(\mathtt{x}), \mathtt{true}, \mathtt{pred}(\underline{\mathtt{if}(\mathtt{false}, \mathtt{0}, \mathtt{if}(\mathtt{?0}(\mathtt{y}), \mathtt{x}, \mathtt{minus}(\mathtt{pred}(\mathtt{x}), \mathtt{pred}(\mathtt{y}))))})=\mathtt{minus}(\mathtt{pred}(\mathtt{x}),\mathtt{y}))$ $\rightarrow$

$\mathtt{if}(\mathtt{?0}(\mathtt{x}), \mathtt{true}, \mathtt{pred}(\mathtt{if}(\underline{\mathtt{?0}(\mathtt{y})}, \mathtt{x}, \mathtt{minus}(\mathtt{pred}(\mathtt{x}), \mathtt{pred}(\mathtt{y}))))=\mathtt{minus}(\mathtt{pred}(\mathtt{x}),\mathtt{y}))$ $\rightarrow$

$\mathtt{if}(\mathtt{?0}(\mathtt{x}), \mathtt{true}, \mathtt{pred}(\underline{\mathtt{if}(\mathtt{true}, \mathtt{x}, \mathtt{minus}(\mathtt{pred}(\mathtt{x}), \mathtt{pred}(\mathtt{y})))})=\mathtt{minus}(\mathtt{pred}(\mathtt{x}),\mathtt{y}))$ $\rightarrow$

$\mathtt{if}(\mathtt{?0}(\mathtt{x}), \mathtt{true}, \mathtt{pred}(\mathtt{x})=\underline{\mathtt{minus}(\mathtt{pred}(\mathtt{x}),\mathtt{y})})$ $\rightarrow$

$\mathtt{if}(\mathtt{?0}(\mathtt{x}), \mathtt{true}, \mathtt{pred}(\mathtt{x})=\mathtt{if}(\mathtt{?0}(\mathtt{pred}(\mathtt{x})), \mathtt{0}, \mathtt{if}(\underline{\mathtt{?0}(\mathtt{y})}, \mathtt{pred}(\mathtt{x}), \mathtt{minus}(\mathtt{pred}(\mathtt{pred}(\mathtt{x})), \mathtt{pred}(\mathtt{y})))))$ $\rightarrow$

$\mathtt{if}(\mathtt{?0}(\mathtt{x}), \mathtt{true}, \mathtt{pred}(\mathtt{x})=\mathtt{if}(\mathtt{?0}(\mathtt{pred}(\mathtt{x})), \mathtt{0}, \underline{\mathtt{if}(\mathtt{true}, \mathtt{pred}(\mathtt{x}), \mathtt{minus}(\mathtt{pred}(\mathtt{pred}(\mathtt{x})), \mathtt{pred}(\mathtt{y})))}))$ $\rightarrow$

$\mathtt{if}(\mathtt{?0}(\mathtt{x}), \mathtt{true}, \underline{\mathtt{pred}(\mathtt{x})=\mathtt{if}(\mathtt{?0}(\mathtt{pred}(\mathtt{x})), \mathtt{0}, \mathtt{pred}(\mathtt{x}))})$ $\rightarrow$

$\mathtt{if}(\mathtt{?0}(\mathtt{x}), \mathtt{true}, \mathtt{if}(\mathtt{?0}(\mathtt{pred}(\mathtt{x})), \underline{\mathtt{pred}(\mathtt{x})=\mathtt{0}}, \mathtt{pred}(\mathtt{x})=\mathtt{pred}(\mathtt{x})))$ $\rightarrow$

$\mathtt{if}(\mathtt{?0}(\mathtt{x}), \mathtt{true}, \mathtt{if}(\mathtt{?0}(\mathtt{pred}(\mathtt{x})), \underline{\mathtt{?0}(\mathtt{pred}(\mathtt{x}))}, \mathtt{pred}(\mathtt{x})=\mathtt{pred}(\mathtt{x})))$ $\rightarrow$

$\mathtt{if}(\mathtt{?0}(\mathtt{x}), \mathtt{true}, \mathtt{if}(\mathtt{?0}(\mathtt{pred}(\mathtt{x})), \mathtt{true}, \underline{\mathtt{pred}(\mathtt{x})=\mathtt{pred}(\mathtt{x})}))$ $\rightarrow$

$\mathtt{if}(\mathtt{?0}(\mathtt{x}), \mathtt{true}, \underline{\mathtt{if}(\mathtt{?0}(\mathtt{pred}(\mathtt{x})), \mathtt{true}, \mathtt{true})})$ $\rightarrow$

$\underline{\mathtt{if}(\mathtt{?0}(\mathtt{x}), \mathtt{true}, \mathtt{true})}$ $\rightarrow$

$\mathtt{true}$

Abbildung 9.5: Symbolische Auswertung des Goal-Terms der Sequenz $\{\mathtt{?0(y)}\}, \emptyset \vdash \mathtt{if(x>y, pred(minus(x,y))=minus(pred(x),y), true)}$.

Die Execution-Guards der Prozedur `minus` und der Prozedur `>` seien wie folgt gegeben:

$$\begin{aligned} exec_{\texttt{>}} &= \texttt{if}(\texttt{?0}(\texttt{x}), \texttt{true}, \texttt{?0}(\texttt{y})), \\ exec_{\texttt{minus}} &= \texttt{if}(\texttt{?0}(\texttt{x}), \texttt{true}, \texttt{?0}(\texttt{y})). \end{aligned}$$

Betrachten wir nun den Beweis des folgenden Lemmas:

```
lemma minus_pred ⇐ all x : nat
  if(x>y, pred(minus(x, y))=minus(pred(x), y), true).
```

Die Teilterme $\texttt{minus}(\texttt{x}, \texttt{y})$ und $\texttt{minus}(\texttt{pred}(\texttt{x}), \texttt{y})$ schlagen zum Beweis dieses Lemmas eine Induktion über R_2 vor. Für diese Induktion erhalten wir als Induktionsanfang die folgende Sequenz

$$\{\texttt{?0}(\texttt{y})\}, \emptyset \vdash \texttt{if}(\texttt{x>y}, \texttt{pred}(\texttt{minus}(\texttt{x}, \texttt{y}))=\texttt{minus}(\texttt{pred}(\texttt{x}), \texttt{y}), \texttt{true}).$$

Es ergibt sich dann die in Abbildung 9.5 dargestellte symbolische Auswertung. Alle Prozeduraufrufe von `>` und `minus` werden hierbei aufgrund der Regel „*Execute procedure call (non recursive cases)*" ausgewertet. □

9.4 Nicht-rekursiv definierte Prozeduren

Kommen wir nun zu nicht-rekursiv definierten Prozeduren. Obwohl die uneingeschränkte Auswertung dieser Prozeduren keine Endlos-Auswertungen erzeugen kann, ist es nicht sinnvoll, nicht-rekursiv definierte Prozeduren auf diese Art zu verwenden. Der Grund hierfür ist, dass wir durch eine uneingeschränkte Auswertung für Prozedurdefinitionen wie

```
function swap(h : tree) : tree ⇐
  if(h=tip,
     tip,
     if(left(h)=tip,
        tip,
        if(depth(left(h))=depth(right(h)),
           node(left(h), bottom(right(h)), pop(right(h)))
           node(pop(left(h)), bottom(left(h)), right(h)))))
```

mit der Typoperatordefinition

```
structure tree ⇐
  tip, node(left : tree, val : nat, right : tree)
```

einen einfachen Prozeduraufruf wie `swap(h)` durch einen entsprechend komplizierten `if`-Term ersetzen und so zusätzliche Fallunterscheidungen in den auszuwertenden Term einführen. Wir definieren daher zunächst eine Regel auf Basis folgender Heuristik:

Heuristik (keine zusätzlichen Fallunterscheidungen):
Ist gewährleistet, dass durch die Auswertung eines Prozeduraufrufs $f(t^*)$ einer nicht-rekursiv definierten Prozedur f keine zusätzlichen Fallunterscheidungen entstehen, so wird der Prozeduraufruf ausgewertet.

Für eine nicht-rekursiv definierte Prozedur bezeichnen wir einen Prozeduraufruf $f(t_1, \ldots, t_n)$ als *trivial*, falls für alle $i \in \{1, \ldots, n\}$ die Argumente t_i Konstruktorgrundterme sind. Triviale Prozeduraufrufe nicht-rekursiv definierter Prozeduren

können immer zu Konstruktorgrundtermen ausgewertet werden. Sie führen somit keine neuen Fallunterscheidungen ein und können somit aufgrund der Heuristik ausgewertet werden. Ähnlich wie für die trivialen Prozeduraufrufe rekursiv-definierter Prozeduren müssen wir für die trivialen Prozeduraufrufe nicht-rekursiv definierter Prozeduren sicherstellen, dass die Auswertung des Prozeduraufrufs unabhängig von den Symbolen $*$ des Rumpfs ist, falls die Prozedur unvollständig-definiert ist. Wir verwenden hierzu wieder das Prädikat $det_{\mathcal{U}}$. Diese Überlegungen führen zu folgender vorläufiger Regel:

$$\frac{f(t_1,\ldots,t_n)}{\sigma_{\xi_{f(\ldots)}}(R_f^*)}, \quad \begin{array}{l}\text{falls } f \in \Sigma \cap \Sigma^{\text{n-rec}}(P),\ \sigma = \{x_1/t_1,\ldots,x_n/t_n\},\\ \mathsf{e}(t_1),\ldots,\mathsf{e}(t_n) \in \mathcal{T}erm^{\text{cons}}(P) \text{ und } det_{\mathcal{U}}(f(\mathsf{e}(t_1),\ldots,\mathsf{e}(t_n))).\end{array} \tag{9.14}$$

Können für den instantiierten `let`-bereinigten Prozedurrumpf $\sigma(|R_f|_p)$ und eine Ergebnisposition π der Prozedur f alle Bedingungen

$$b \in cond(\sigma(\pi, |R_f|_p)))$$

zu `true` ausgewertet werden, so wissen wir, dass sich $\sigma(|R_f|_p)$ zu dem entsprechenden Ergebnisterm $|R_f|_p|_\pi$ vereinfachen lässt. Für den instantiierten Prozedurrumpf $\sigma(R_f^*)$ bedeutet dies, dass wir ihn zu einem entsprechenden `case`-freien Term auswerten können. Die Auswertung des Prozeduraufrufs führt damit keine Fallunterscheidungen ein und wir erweitern daher Definition (9.14) zu folgender Regel:

47. Execute procedure call (no additional case analysis)

$$\frac{f^{\,A}(t_1,\ldots,t_n)}{\sigma_{\xi_{f(\ldots)}}(\mathsf{s}_f^*(\mathsf{un}_f^{u,s,0}(R_f)))},$$

	Label	S-Lim.	U-Lim.
A:	l	s	u

falls $f \in \Sigma \cap \Sigma^{\text{n-rec}}(P)$, $\sigma = \{x_1/\mathsf{a}_{A_0}(t_1),\ldots,x_n/\mathsf{a}_{A_0}(t_n)\}$ und eine der folgenden Bedingungen erfüllt ist:

(1) Es gilt $\mathsf{e}(t_1),\ldots,\mathsf{e}(t_n) \in \mathcal{T}erm^{\text{cons}}(P)$ und $det_{\mathcal{U}}(f^{\,A}(\mathsf{e}(t_1),\ldots,\mathsf{e}(t_n)))$.

(2) Es gilt $l \notin C$ und es existiert ein $\pi \in \mathcal{P}os_{\text{E}}(f)$, so dass für alle $b \in cond(\pi, |R_f|_p[x_1/\mathsf{e}(t_1),\ldots,x_n/\mathsf{e}(t_n)])$ gilt $\mathcal{U} \vdash \mathsf{a}_{A_{\text{top}}}(b) \to \mathtt{true}$.

Anmerkung 9.19. Die Annotationen des Terms $\sigma_{\xi_{f(\ldots)}}(\mathsf{s}_f^*(\mathsf{un}_f^{u,s,0}(R_f)))$ haben wir wieder aus der Regel „*Execute procedure call (recursive cases)*“ übernommen. Die Annotationen sind aus denselben Gründen sinnvoll bzw. relevant wie für Regel „*Execute procedure call (recursive cases)*“. Lediglich die Labels für rekursive Aufrufe sind irrelevant, da wir nicht-rekursiv definierte Prozeduren betrachten. □

Für eine befriedigende symbolische Auswertung kommen wir nicht umhin, nicht-rekursiv definierte Prozeduren auch dann auszuwerten, wenn dadurch zusätzliche Fallunterscheidungen eingeführt werden. Wir definieren hierzu die folgende Regel:

51. Execute procedure call (additional case analysis)

$$\frac{f^{\,A}(t_1,\ldots,t_n)}{\sigma_{\xi_{f(\ldots)}}(\mathsf{s}^*_f(\mathsf{un}^{u,s,0}_f(R^*_f)))},$$

	Label	S-Lim.	U-Lim.
A:	l	s	u

falls $f \in \Sigma \cap \Sigma^{\text{n-rec}}(P)$, $l \notin C$, $\sigma = \{x_1/\mathsf{a}_{A_0}(t_1), \ldots, x_n/\mathsf{a}_{A_0}(t_n)\}$ und ein Bedingung $b \in \mathit{cond}(\sigma(|R_f|_p))$ existiert mit $(\mathsf{a}_{A_{\text{top}}}(b)\downarrow_{\mathcal{U}'}) \in \{\texttt{true}, \texttt{false}\}$.[8] Hierbei bezeichnet $\mathcal{U}'$ die folgende A-Umgebung:

	Hyp_L
$\mathcal{U}'$:	$\emptyset$

Diese Regel implementiert die folgende Heuristik:

Heuristik (zusätzliche Fallunterscheidungen):
Kann für einen Prozeduraufruf $f(t^*)$ einer nicht-rekursive-definierten Prozedur f die Fallunterscheidung des instantiierten Rumpfs $R_f[x^*/t^*]$ aufgrund der globalen Hypothesen vereinfacht werden, so wird der Prozeduraufruf ausgewertet.

Anmerkung 9.20. Die Regeln zur Auswertung nicht-rekursiv definierter Prozeduren basieren auf einer relativ kleinen Menge von Fallstudien, da die meisten unserer Fallstudien ohne nicht-rekursiv definierte Prozeduren auskommen. In den Fallstudien mit nicht-rekursiv definierten Prozeduren haben sich die Regeln jedoch bewährt. □

Anmerkung 9.21. Zusätzlich zu der Regel „*Execute procedure call (additional case analysis)*“ bzw. als Alternative ist es unter Umständen sinnvoll – ähnlich wie bei den nachfolgend definierten „*Unfold*“-Regeln – den Termkontext $C[\ldots]$ in dem ein Prozeduraufruf $f(t_1,\ldots,t_n)$ auftaucht in die Analyse mit einzubeziehen. So könnte man beispielsweise die folgende Regel definieren:

$$\frac{f^{\,A}(t_1,\ldots,t_n)\texttt{=}r}{\sigma_{\xi_{f(\ldots)}}(\mathsf{s}^*_f(\mathsf{un}^{u,s,0}_f(R^*_f)))\texttt{=}r},$$

	Label	S-Lim.	U-Lim.
A:	l	s	u

falls $f \in \Sigma \cap \Sigma^{\text{n-rec}}(P)$, $l \notin C$, $\sigma = \{x_1/\mathsf{a}_{A_0}(t_1), \ldots, x_n/\mathsf{a}_{A_0}(t_n)\}$ und $\mathsf{a}_{A_{\text{top}}}(R^*_f[x_1/\mathsf{e}(t_1), \ldots, x_n/\mathsf{e}(t_n)])\texttt{=}r)\downarrow_{\mathcal{U}}$ ein `case`-freier Term ist.

Der Vorteil einer solchen Regel im Vergleich zur Regel „*Execute procedure call (additional case analysis)*“ ist, dass diese Regel sicherstellt, dass durch die Auswertung der nicht-rekursiv definierten Prozeduren keine zusätzlichen Fallunterscheidungen eingeführt werden und sich somit die Komplexität des auszuwertenden Terms nicht erhöht. Der Nachteil im Vergleich zur Regel „*Execute procedure call (additional case analysis)*“ ist, dass die Überprüfung der Anwendungsbedingungen teurer ist und es unklar ist, ob alle für uns relevanten Prozeduraufrufe auf diese Weise ausgewertet werden können. □

[8] Mit $\mathit{cond}(\sigma(|R_f|_p))$ bezeichnen wir die Menge $\bigcup_{\pi \in \mathcal{P}os_{\text{E}}(f)} \{|b| \mid b \in \mathit{cond}(\pi, \sigma(|R_f|_p))\}$.

9.5 Kommutative Prozeduren

Betrachten wir zunächst das folgende Beispiel.

Beispiel 9.22. Sei P das Programm $\langle D_{\texttt{plus}}, D_{\texttt{half}}\rangle$, wobei $D_{\texttt{plus}}$ und $D_{\texttt{half}}$ die entsprechenden Prozedurdefinitionen aus Abbildung 8.2 bezeichnen. Weiter seien die Mengen der generalisierten Relationenbeschreibungen gegeben durch $grds(\texttt{plus}) = \{R_1\}$ und $grds(\texttt{half}) = \{R_2\}$, wobei gilt

$$\begin{aligned} R_1 &= \{\langle\langle \neg\texttt{?0(x)}\rangle, \{\{\texttt{x/pred(x)}\}\}\rangle\}, \\ R_2 &= \{\langle\langle \neg\texttt{?0(x)}, \neg\texttt{?0(pred(x))}\rangle, \{\{\texttt{x/pred(pred(x))}\}\}\rangle\}. \end{aligned}$$

Betrachten wir nun den Beweis des folgenden Lemmas:

$$\begin{array}{l}\texttt{lemma plus\_half} \Leftarrow \texttt{all x : nat} \\ \quad \texttt{half(plus(x,x))=x.}\end{array}$$

Zum Beweis des Lemmas schlägt der Teilterm $\texttt{plus(x,x)}$ eine Induktion über die Relationenbeschreibung R_1 vor. Für diese Induktion erhalten wir als Induktionsschritt die folgende Sequenz:

$$\{\neg\texttt{?0(x)}\}, \{\texttt{half(plus(pred(x),pred(x)))=pred(x)}\} \vdash \texttt{half(plus(x,x))=x.} \quad (9.15)$$

Für diese Sequenz ergibt sich dann die folgende symbolische Auswertung:

$$\begin{array}{lll} \mathcal{U} \vdash & \texttt{half}(\underline{\texttt{plus(x,x)}}) & \rightarrow \\ & \texttt{half}(\texttt{if}(\underline{\texttt{?0(x)}}, \texttt{x}, \texttt{succ}(\texttt{plus}(\texttt{pred(x)}, \texttt{x})))) & \rightarrow \\ & \texttt{half}(\underline{\texttt{if}(\texttt{false}, \texttt{x}, \texttt{succ}(\texttt{plus}(\texttt{pred(x)}, \texttt{x})))}) & \rightarrow \\ & \texttt{half}(\texttt{succ}(\texttt{plus}(\texttt{pred(x)}, \texttt{x}))). & \end{array}$$

Der Term $\texttt{half(succ(plus(pred(x),x)))=x}$ ist nicht weiter auswertbar. Da wir auf diesen Term die Induktionshypothese offensichtlich nicht anwenden können, haben wir die Gültigkeit der Sequenz (9.15) durch die symbolische Auswertung nicht bewiesen. Für die Prozedur `plus` gilt jedoch die folgende Quantifikation:

$$\texttt{all x : nat, y : nat plus(x,y)=plus(y,x).}$$

Das bedeutet, dass `plus` kommutativ ist. Ist im ✓`eriFun`-System ein entsprechendes Lemma verifiziert, so können wir die Argumente des `plus`-Terms im Ergebnis der symbolischen Auswertung vertauschen. Es ergibt sich dann der folgende Term:

$$\texttt{half(succ(plus(x,pred(x))))=x.} \quad (9.16)$$

Für den Teilterm $\texttt{plus(x,pred(x))}$ ist die Regel „*Execute procedure call (recursive cases)*“ anwendbar und wir erhalten für den Term (9.16) die folgende symbolische Auswertung:

$\mathcal{U} \vdash$ half(succ(plus(x, pred(x))))=x →
half(succ(if(?0(x), x, succ(plus(pred(x), pred(x))))))=x →
half(succ(if(false, x, succ(plus(pred(x), pred(x))))))=x →
half(succ(succ(plus(pred(x), pred(x)))))=x →
if(?0(succ(succ(plus(pred(x), pred(x))))), . . . , if(. . . , . . . , . . .))=x →
if(false, . . . , if(?0(pred(succ(succ(. . .)))), . . . , . . .))=x →
if(?0(pred(succ(succ(plus(pred(x), pred(x)))))), . . . , . . .)=x →
if(?0(succ(plus(pred(x), pred(x)))), . . . , . . .)=x →
if(false, . . . , succ(half(pred(pred(succ(succ(. . .)))))))=x →
succ(half(pred(pred(succ(succ(plus(pred(x), pred(x)))))))))=x →
succ(half(pred(succ(plus(pred(x), pred(x))))))=x →
succ(half(plus(pred(x), pred(x))))=x →
succ(half(plus(pred(x), pred(x))))=succ(pred(x)) →
half(plus(pred(x), pred(x)))=pred(x).

Das Ergebnis half(plus(pred(x), pred(x)))=pred(x) dieser symbolischen Auswertung stimmt offensichtlich mit der Induktionshypothese der Sequenz (9.15) überein. Wir haben damit, durch die Verwendung der Kommutativität von plus, die Gültigkeit der Sequenz bewiesen. □

Das Beispiel zeigt, dass es für die Anwendung der Regel „*Execute procedure call (recursive cases)*“ sinnvoll sein kann, die Argumente von Prozeduraufrufen kommutativer Prozeduren zu vertauschen, da erst durch diese Vertauschung die Anwendungsbedingungen der Regel erfüllt sind. Ähnliche Beispiele lassen sich auch für die restlichen „*Execute*“-Regeln angeben. Wir definieren daher für jede dieser Regeln eine „kommutierte“ Version, die überprüft, ob für einen Prozeduraufruf $f(t_1, t_2)$ einer kommutativen Prozedur f die Regel nach Vertauschen der Argumente anwendbar ist, und falls dies der Fall ist, den Prozeduraufruf mit entsprechend vertauschten Argumenten t_2 und t_1 auswertet. Wir verwenden also zur Definition der kommutierten Versionen der Regeln die folgende Heuristik

Heuristik (Auswertung kommutativer Prozeduren):
Ist die Prozedur f kommutativ und kann der Prozeduraufruf $f(t_1, t_2)$ nach vertauschen der Argumente t_1 und t_2 mit Hilfe einer der „*Execute*“-Regeln ausgewertet werden, so vertauschen wir die Argumente des Prozeduraufrufs und wenden die entsprechende „*Execute*“-Regel an.

Welche Prozeduren während der symbolischen Auswertung als kommutativ vorausgesetzt werden dürfen, wird durch die Termsignatur Σ_C der aktuellen A-Umgebung festgelegt. Diese Signatur enthält alle Prozeduren, für die in ✓eriFun ein entsprechendes Lemma verifiziert wurde. Es ist jedoch dabei zu beachten, dass für einige kommutative Prozeduren die Verwendung der kommutierten Version der Regel „*Execute procedure call (recursive cases)*“ zu Endlos-Auswertungen führen kann. Betrachten wir dazu ein Beispiel.

Beispiel 9.23. Sei P das Programm $\langle D_{\mathtt{plus}'}\rangle$, wobei $D_{\mathtt{plus}'}$ die folgende Prozedurdefinition bezeichnet:

$$\begin{aligned} D_{\mathtt{plus}'} = \mathtt{function}\ \mathtt{plus}'(\mathtt{x}:\mathtt{nat},\mathtt{y}:\mathtt{nat}):\mathtt{nat} \Leftarrow \\ \mathtt{if}(?\mathtt{0}(\mathtt{x}), \\ \quad \mathtt{y}, \\ \quad \mathtt{plus}'(\mathtt{pred}(\mathtt{x}),\mathtt{succ}(\mathtt{y}))) . \end{aligned}$$

Sei weiter die Menge der generalisierten Relationenbeschreibungen gegeben durch $grds(\mathtt{plus}) = \{R\}$, wobei gilt:

$$R \quad = \quad \{\langle\langle \neg ?\mathtt{0}(\mathtt{x})\rangle, \{\{\mathtt{x}/\mathtt{pred}(\mathtt{x})\}\}\rangle\} .$$

Die Prozedur $\mathtt{plus}'$ ist offensichtlich kommutativ. Betrachten wir nun die symbolische Auswertung des Terms $\mathtt{plus}'(\mathtt{succ}(\mathtt{v}),\mathtt{w})$:

$$\begin{aligned} \mathcal{U} \vdash\ & \underline{\mathtt{plus}'(\mathtt{succ}(\mathtt{v}),\mathtt{w})} && \rightarrow \\ & \mathtt{if}(\underline{?\mathtt{0}(\mathtt{succ}(\mathtt{v}))},\mathtt{w},\mathtt{plus}'(\mathtt{pred}(\mathtt{succ}(\mathtt{v})),\mathtt{succ}(\mathtt{w}))) && \rightarrow \\ & \underline{\mathtt{if}(\mathtt{false},\mathtt{w},\mathtt{plus}'(\mathtt{pred}(\mathtt{succ}(\mathtt{v})),\mathtt{succ}(\mathtt{w})))} && \rightarrow \\ & \mathtt{plus}'(\underline{\mathtt{pred}(\mathtt{succ}(\mathtt{v}))},\mathtt{succ}(\mathtt{w})) && \rightarrow \\ & \mathtt{plus}'(\mathtt{v},\mathtt{succ}(\mathtt{w})). \end{aligned}$$

Auf den Term $\mathtt{plus}'(\mathtt{v},\mathtt{succ}(\mathtt{w}))$ können wir die kommutierte Version der Regel „*Execute procedure call (recursive cases)*“ anwenden, da nach Vertauschen der Argumente des Terms $\mathtt{plus}'(\mathtt{v},\mathtt{succ}(\mathtt{w}))$ die Regel „*Execute procedure call (recursive cases)*“ anwendbar ist. Eine Anwendung der kommutierten Version der Regel liefert den Term

$$\mathtt{if}(?\mathtt{0}(\mathtt{succ}(\mathtt{w})),\mathtt{v},\mathtt{plus}'(\mathtt{pred}(\mathtt{succ}(\mathtt{w})),\mathtt{succ}(\mathtt{v}))) .$$

Dieser Term kann dann durch folgende symbolische Auswertung weiter vereinfacht werden:

$$\begin{aligned} \mathcal{U} \vdash\ & \mathtt{if}(\underline{?\mathtt{0}(\mathtt{succ}(\mathtt{w}))},\mathtt{v},\mathtt{plus}'(\mathtt{pred}(\mathtt{succ}(\mathtt{w})),\mathtt{succ}(\mathtt{v}))) && \rightarrow \\ & \underline{\mathtt{if}(\mathtt{false},\mathtt{v},\mathtt{plus}'(\mathtt{pred}(\mathtt{succ}(\mathtt{w})),\mathtt{succ}(\mathtt{v})))} && \rightarrow \\ & \mathtt{plus}'(\underline{\mathtt{pred}(\mathtt{succ}(\mathtt{w}))},\mathtt{succ}(\mathtt{v})) && \rightarrow \\ & \mathtt{plus}'(\mathtt{w},\mathtt{succ}(\mathtt{v})). \end{aligned}$$

Für den Term $\mathtt{plus}'(\mathtt{w},\mathtt{succ}(\mathtt{v}))$ ist dann wieder die kommutierte Version der Regel „*Execute procedure call (recursive cases)*“ anwendbar, wodurch wir den Prozeduraufruf $\mathtt{plus}'(\mathtt{v},\mathtt{succ}(\mathtt{w}))$ erhalten und damit in eine Endlos-Auswertung geraten. □

Die Ursache für die Endlos-Auswertung in Beispiel 9.23 ist in der Prozedurdefinition von $\mathtt{plus}'$ zu finden. Für den rekursiven Aufruf $\mathtt{plus}'(\mathtt{pred}(\mathtt{x}),\mathtt{succ}(\mathtt{y}))$ in der Prozedurdefinition $D_{\mathtt{plus}'}$ ergibt sich durch Vertauschen der Argumente der Prozeduraufruf $\mathtt{plus}'(\mathtt{succ}(\mathtt{y}),\mathtt{pred}(\mathtt{x}))$. Für diesen Prozeduraufruf kann die Regel „*Execute procedure call (recursive cases)*“ unabhängig von denen für $\mathtt{x}$ und $\mathtt{y}$ eingesetzten Termen angewendet werden. Wir stellen daher die folgende Anforderung:

Anforderung (kommutative Auswertung)
Für alle rekursiven Aufrufe $f(t_1, t_2)$ einer kommutativen Prozedur f im Rumpf R_f dürfen nach Vertauschung der Argumente t_1 und t_2 die Anwendungsbedingungen der Regel „*Execute procedure call (recursive cases)*“ nicht erfüllt sein. Ist dies nicht der Fall, so darf die kommutierte Version der Regel „*Execute procedure call (recursive cases)*“ nicht für die Prozedur f verwendet werden.

Zur Überprüfung dieser Forderung definieren wir folgendes Prädikat:

$$\textit{comm-exec}_P^{\Sigma_C}(f)$$
$$:\Longleftrightarrow$$

$\forall \pi \in \mathcal{Pos}(|R_f|_{\mathcal{P}})$ mit $|R_f|_{\mathcal{P}}|_\pi = f(t_1, t_2)$ und $\forall \langle D_1, \Delta_1 \rangle \in \bigcup_{R \in grds(f)} R$ mit $D_1 \subseteq \mathit{cond}(\pi, |R_f|_{\mathcal{P}}|)$ sowie $\forall \langle D_2, \Delta_2 \rangle \in \bigcup_{R \in grds(f)} R$ gilt

$$\left.\begin{array}{l} \mathcal{U} \vdash \mathtt{a}_{A_{\text{std}}}(\mathit{AND}(D_2[x_1/t_2, x_2/t_1]) \not\rightarrow^! \mathtt{true}\,. \\ \text{Hierbei bezeichnet } \mathcal{U} \text{ die folgende A-Umgebung:} \end{array}\right\} (*)$$

$\mathcal{U}_0$	Prog.	Proz.	Komm.	Hyp_G
$\mathcal{U}$:	P	$\Sigma^{\text{proc}}(P)$	Σ_C	D_1

Zur Erklärung des Prädikates $\textit{comm-exec}_P^{\Sigma_C}$:

Der mit $(*)$ markierte Teil der Definition repräsentiert die Aussage

$$P \models \mathtt{all}\ \nu^*, x^* : \tau^*\ \mathtt{if}(\mathit{AND}(D_1), \neg(\mathit{AND}(D_2[x_1/t_2, x_2/t_1])), \mathtt{true}).$$

Die Terme D_1 stellen hierbei die Bedingungen dar, unter denen der rekursive Aufruf $|R_f|_{\mathcal{P}}\big|_\pi$ ausgewertet wird. Sind diese Bedingungen nicht erfüllt, so wird der rekursive Aufruf $|R_f|_{\mathcal{P}}\big|_\pi$ während einer entsprechenden symbolischen Auswertung nicht betrachtet und die Vertauschung der Argumente t_1 und t_2 kann demzufolge auch nicht zu einer Endlos-Auswertung führen. Das Prädikat muss daher lediglich gewährleisten, dass unter der Voraussetzung, dass die Bedingungen D_1 erfüllt sind, durch die Vertauschung der Argumente des rekursiven Aufrufs $|R_f|_{\mathcal{P}}\big|_\pi$ die Anwendungsbedingung der Regel „*Execute procedure call (recursive cases)*“ nicht erfüllt ist.

Die Verwendung der Bedingungen D_1 als globale Hypothesen in $(*)$ ist notwendig, da diese Bedingungen während einer symbolischen Auswertung eines Induktionsanfangs oder Induktionsschritts typischerweise als globale Hypothesen auftreten. Die Auswertung der Prozeduren ist notwendig, da diese in der Anwendungsbedingung der Regel „*Execute procedure call (recursive cases)*“ ebenfalls ausgewertet werden dürfen.

Wir verhindern die Anwendung der kommutierten Version der Regel „*Execute procedure call (recursive cases)*“ für einen Prozeduraufruf $f(t_1, t_2)$, falls für f das Prädikat $\textit{comm-exec}_P^{\Sigma_C}$ nicht erfüllt ist. Insgesamt ergeben sich die folgenden Regeldefinitionen.

42. Execute procedure call (constructor ground terms)*

$$\frac{f^{\,A}(t_1,t_2)}{\sigma_{\xi_{f(\ldots)}}(\mathsf{s}^*_f(\mathsf{un}^{u,s,0}_f(R^*_f)))},$$

	S-Lim.	U-Lim.
A:	s	u

falls $f \in \Sigma \cap \Sigma_{\mathrm{C}} \cap \Sigma^{\mathrm{rec}}(P)$, $\sigma = \{x_1/\mathsf{a}_{A_0}(t_2), x_2/\mathsf{a}_{A_0}(t_1)\}$ gilt und eine Relationenbeschreibung $R \in \mathit{grds}_P(f)$ existiert mit $\mathsf{e}(\sigma(x)) \in \mathcal{T}\mathit{erm}^{\mathrm{cons}}(P)$ für alle $x \in \mathit{iv}(R)$. Weiterhin muss gelten $\mathit{det}_{\mathcal{U}}(\mathsf{e}(f^{\,A}(t_2,t_1)))$.

44. Execute procedure call (recursive cases)*

$$\frac{f^{\,A}(t_1,t_2)}{\sigma_{\xi_{f(\ldots)}}(\mathsf{s}^*_f(\mathsf{un}^{u,s,0}_f(R_f)))},$$

	Label	S-Lim.	U-Lim.
A:	l	s	u

falls $f \in \Sigma \cap \Sigma_{\mathrm{C}}\Sigma^{\mathrm{rec}}(P)$, $l \notin C$, $\mathit{comm\text{-}exec}^{\Sigma_{\mathrm{C}}}_P(f)$, $\sigma = \{x_1/\mathsf{a}_{A_0}(t_2), x_2/\mathsf{a}_{A_0}(t_1)\}$ und eine Relationenbeschreibung $R \in \mathit{grds}_P(f)$ und eine atomare Relationenbeschreibung $\langle D, \Delta\rangle \in R$ existieren mit

$$\mathcal{U}' \vdash \mathsf{a}_{A_{\mathrm{std}}}(\mathit{AND}(D))[x_1/\mathsf{a}_{A_{\mathrm{top}}}(t_2), x_2/\mathsf{a}_{A_{\mathrm{top}}}(t_1)] \to^{!} \mathtt{true}.$$

Hierbei bezeichnet $\mathcal{U}'$ die folgende A-Umgebung:

	$\mathrm{Hyp}_{\mathrm{L}}$	Termsig.
$\mathcal{U}'$:	$\emptyset$	$\Sigma \setminus \{f\}$

46. Execute procedure call (non recursive cases)*

$$\frac{f^{\,A}(t_1,t_2)}{\sigma_{\xi_{f(\ldots)}}(\mathsf{s}^*_f(\mathsf{un}^{u,s,0}_f(R_f)))},$$

	Label	S-Lim.	U-Lim.
A:	l	s	u

falls $f \in \Sigma \cap \Sigma_{\mathrm{C}} \cap \Sigma^{\mathrm{rec}}(P)$, $l \notin C$, $\sigma = \{x_1/\mathsf{a}_{A_0}(t_2), x_2/\mathsf{a}_{A_0}(t_1)\}$ und es gilt

$$\mathcal{U} \vdash \mathsf{a}_{A_{\mathrm{top}}}(\mathit{exec}_f[x_1/\mathsf{e}(t_2), x_2/\mathsf{e}(t_1)]) \to^{!} \mathtt{true}\,.$$

48. Execute procedure call (no additional case analysis)*

$$\frac{f^{\,A}(t_1,t_2)}{\sigma_{\xi_{f(\ldots)}}(\mathsf{s}^*_f(\mathsf{un}^{u,s,0}_f(R_f)))},$$

	Label	S-Lim.	U-Lim.
A:	l	s	u

falls $f \in \Sigma \cap \Sigma_{\mathrm{C}} \cap \Sigma^{\text{n-rec}}(P)$, $\sigma = \{x_1/\mathsf{a}_{A_0}(t_2), x_2/\mathsf{a}_{A_0}(t_1)\}$, $l \notin C$ und ein $\pi \in \mathcal{P}\mathit{os}_{\mathrm{E}}(f)$ existiert, so dass für alle $b \in \mathit{cond}(\pi, \sigma(|R_f|_{\mathcal{P}}))$ gilt $\mathcal{U} \vdash \mathsf{a}_{A_{\mathrm{top}}}(b) \to \mathtt{true}$.

52. Execute procedure call (additional case analysis)*

$$\frac{f^{\,A}(t_1, t_2)}{\sigma_{\xi_{f(\ldots)}}(\mathbf{s}^*_f(\mathbf{un}^{u,s,0}_f(R_f)))},$$

	Label	S-Lim.	U-Lim.
A:	l	s	u

falls $f \in \Sigma \cap \Sigma_{\mathrm{C}} \cap \Sigma^{\text{n-rec}}(P)$, $l \notin C$, $\sigma = \{x_1/\mathbf{a}_{A_0}(t_2), x_2/\mathbf{a}_{A_0}(t_1)\}$ und ein Bedingung $b \in \mathit{cond}(\sigma(|R|_{\mathcal{P}}))$ existieren mit $(\mathbf{a}_{A_{\mathrm{top}}}(b)\downarrow_{\mathcal{U}'}) \in \{\mathtt{true}, \mathtt{false}\}$. Hierbei bezeichnet $\mathcal{U}'$ die folgende A-Umgebung:

	$\mathrm{Hyp}_{\mathrm{L}}$
$\mathcal{U}'$:	$\emptyset$

9.6 Abschließende Anmerkungen

Die in diesem Kapitel definierten „*Execute*“-Regeln sind nach steigenden Kosten bei Überprüfung der Anwendungsbedingungen angeordnet. Diese Reihenfolge der Regeln ist für die Effizienz des Auswertungskalküls von Bedeutung. Zur Implementierung des Auswertungskalküls im ✓eriFun-System ist Folgendes anzumerken:

- Die „*Execute*“-Regeln werden in der Implementierung des ✓eriFun-Systems zu einer einzigen Regel zusammengefasst. Dies ist der Grund, warum wir für alle „*Execute*“-Regeln die Ergebnisterme gleich annotieren. Durch diese gleichen Annotationen definieren nämlich alle „*Execute*“-Regeln die gleiche Ersetzungsregel, wobei lediglich verschiedene Anwendungsbedingungen definiert werden. Die im ✓eriFun-System implementierte Regel wird „*Execute procedure call*“ genannt.

- Die Implementierung der kommutierten Versionen der „*Execute*“-Regeln im ✓eriFun-System weicht insofern von unserer Definition ab, da die Argumente der rekursiven Prozeduraufrufe im Rumpf $\sigma(\mathbf{s}^*_f(\mathbf{un}^{u,s}_f(R_f)))$ im ✓eriFun-System zusätzlich vertauscht werden. Dies hat zur Folge, dass ein Prozeduraufruf wie $\mathtt{plus}(\mathtt{x}, \mathtt{succ}(\mathtt{y}))$ in ✓eriFun zu $\mathtt{succ}(\mathtt{plus}(\mathtt{x}, \mathtt{y}))$ ausgewertet wird und nicht, wie durch unsere Regeldefinitionen vorgegeben, zu $\mathtt{succ}(\mathtt{plus}(\mathtt{y}, \mathtt{x}))$. Diese zusätzliche Vertauschung hat lediglich „optische“ Gründe und hat für die weitere Auswertung und Verwendung der Terme keine Konsequenzen.

Kapitel 10

Die „*Unfold*“-Regeln

Wie wir bereits in Abschnitt 6.1 beschrieben haben, existieren zwei Klassen von Regeln zur Auswertung von Prozeduraufrufen. Die erste Klasse von Regeln versucht mit Hilfe verschiedener Heuristiken vor einer Auswertung des Prozeduraufrufs $\sigma(f(x_1, \ldots, x_n))$ zu ermitteln, ob die Auswertung sinnvoll ist. Falls die Heuristiken eine Auswertung für gut befinden, so ersetzen die Regeln den Prozeduraufruf durch den entsprechend instantiierten Prozedurrumpf $\sigma_{\xi_{f(\ldots)}}(R_f^*)$. Diese Klasse von Regeln wird durch die „*Execute*“-Regeln realisiert. Häufig ist es jedoch nicht möglich vor einer Auswertung eines Prozeduraufrufs zu beurteilen, ob eine Auswertung sinnvoll ist. Aus diesem Grund definieren wir eine zweite Klasse von Regeln zur Auswertung von Prozeduraufrufen. Die Regeln dieser Klasse werten zunächst einen Prozeduraufruf $\sigma(f(x_1, \ldots, x_n))$ probeweise vollständig aus und überprüfen anschließend, ob die Auswertung sinnvoll war. Hierbei wird der Termkontext, in dem der Prozeduraufruf enthalten ist, in die Auswertung mit einbezogen.[1] Als sinnvoll wird eine Auswertung erachtet, falls alle rekursiven Prozeduraufrufe von f durch die Auswertung entfernt werden können. Ist dies der Fall, so ersetzen auch die Regeln der zweiten Klasse den Prozeduraufruf durch den entsprechend instantiierten Prozedurrumpf $\sigma_{\xi_{f(\ldots)}}(R_f^*)$. Die Regeln arbeiten somit auf Basis der folgenden Heuristik:

Heuristik (Prozeduraufrufe in Termkontexten):
Können für einen Prozeduraufrufe $C[f(t_1, \ldots, t_n)]$ in einem Termkontext $C[\ldots]$ durch symbolische Auswertung des Terms

$$C[R_f^*[x_1/t_1, \ldots, x_1/t_n]]$$

alle rekursiven Prozeduraufrufe von f eliminiert werden, so ist eine Auswertung des Prozeduraufrufs innerhalb des Termkontexts sinnvoll und der Prozeduraufruf wird durch den instantiierten Prozedurrumpf ersetzt.

Die zweite Klasse von Regeln wird durch die so genannten „*Unfold*“-Regeln realisiert. Betrachten wir vor Definition dieser Regel ein Beispiel:

Beispiel 10.1. Sei P das Programm $\langle D_{\texttt{plus}} \rangle$ wobei $D_{\texttt{plus}}$ die Prozedurdefinition aus Abschnitt 8.2 bezeichnet. Der Term `?0(plus(x,y))` ist mit unseren aktuell definierten Regeln unter einer leeren Hypothesenmenge nicht auswertbar. Nehmen wir jedoch an, dass wir den Prozeduraufruf von `plus` trotzdem auswerten können, so ergibt sich die folgende Auswertung:

[1] Als Termkontext eines Prozeduraufrufs bezeichnen wir die den Prozeduraufruf umgebende Termstruktur. Beispielsweise bezeichnen wir für den Term `?0(plus(x,y))` die Termstruktur `?0(...)` als Termkontext des Prozeduraufrufs `plus(x,y)`.

$$\begin{aligned}
\mathcal{U} \vdash\ & \underline{\texttt{?0(plus(x,y))}} && \rightarrow \\
& \underline{\texttt{?0(if(?0(x),y,succ(plus(pred(x),y))))}} && \rightarrow \\
& \texttt{if(?0(x),?0(y),}\underline{\texttt{?0(succ(plus(pred(x),y))))}}\texttt{)} && \rightarrow \\
& \texttt{if(?0(x),?0(y),false)}. &&
\end{aligned}$$

Durch die Auswertung des Prozeduraufrufs können wir also den instantiierten Prozedurrumpf zu einem Term vereinfachen, der keinen rekursiven Prozeduraufrufe von `plus` mehr enthält. Die Auswertung ist also sinnvoll. Betrachten wir die Auswertung genauer, so erkennen wir, dass für das Verschwinden des rekursiven Aufrufs $\texttt{plus(pred(x),y)}$ der Termkontext $\texttt{?0}(\ldots)$ entscheidend ist. Dieser Termkontext wird im zweiten Auswertungsschritt durch die Regel „*Distribute argument*“ in den instantiierten Prozedurrumpf hineingezogen und führt dann im nächsten Auswertungsschritt dafür dazu, dass durch Anwendung der Regel „*Negative structure test*“ der rekursive Aufruf entfernt werden kann. □

Für Prozeduraufrufe sind insbesondere Termkontexte der Form

$$\begin{gathered}
\texttt{?}\mathit{cons}(\ldots), \\
\mathit{sel}(\ldots), \\
\mathit{l}\texttt{=}\ldots, \\
\ldots\texttt{=}\mathit{r}, \\
\texttt{case}(\ldots, \mathit{cons}^* : t^*)
\end{gathered} \tag{10.1}$$

interessant, da für die entsprechenden Funktionssymbole $\texttt{?}\mathit{cons}$, sel, `=` und `case` spezielle Auswertungsregeln definiert wurden. Für jeden der Termkontexte in (10.1) und für einige Kombinationen dieser Termkontexte definieren wir daher nachfolgend „*Unfold*“-Regeln und unterstützen so die Auswertung von Prozeduraufrufen innerhalb dieser Termkontexte. Konkret definieren wir die folgenden Regeln:

⋮	⋮
63.	*Unfold structure predicate argument*
65.	*Unfold selector argument*
67.	*Unfold case condition*
69.	*Unfold left equality argument*
71.	*Unfold right equality argument*
73.	*Unfold structure predicate selector argument*
75.	*Unfold left equality selector argument*
77.	*Unfold right equality selector argument*
⋮	⋮

Für Prozeduraufrufe, die in keinem der in (10.1) genannten Termkontexte enthalten sind, kann es ebenfalls sinnvoll sein den Prozeduraufruf probeweise vollständig auszuwerten. Wir definieren daher zusätzlich eine „*Unfold*“-Regel, die einen Prozeduraufruf *unabhängig* vom Termkontext auswertet:

⋮	⋮
53.	*Unfold procedure call*
⋮	⋮

Für jede „*Unfold*“-Regeln führen wir außerdem eine kommutierte Version ein, um die Auswertung von Prozeduraufrufen kommutativer Prozeduren zu unterstützen.

Zur Definition der „*Unfold*"-Regeln gehen wir nun wie folgt vor. In Abschnitt 10.1 führen wir die Auswertungsregeln für die Termkontexte „$?cons(\ldots)$" und „$sel(\ldots)$" ein. Hierzu definieren wir unter anderem, wie die Annotationen des instantiierten Prozedurrumpfs zu setzen sind, um eine effiziente symbolische Auswertung des instantiierten Rumpfs zu gewährleisten. Anschließend führen wir in Abschnitt 10.2 die Regeln zur Auswertung von Prozeduraufrufen in Gleichungen und `case`-Termen ein. Auf die Regeln zur Berücksichtigung zusammengesetzter Termkontexte gehen wir in Abschnitt 10.3 ein. Danach definieren wir in Abschnitt 10.4 die Regel „*Unfold procedure call*". Abschließend gehen wir dann in Abschnitt 10.5 auf die kommutierten Versionen der Regeln ein sowie auf die Implementierung der Regeln durch das ✓`eriFun`-System ein.

10.1 Regeln für Strukturprädikate und Selektoren

Wir definieren in diesen Abschnitt die Regeln „*Unfold structure predicate argument*" und „*Unfold selector argument*" zur Auswertung von Prozeduraufrufen, die von einem Strukturprädikat bzw. einem Selektor umgeben sind. Hierzu gehen wir wie folgt vor. In Abschnitt 10.1.1 geben wir zunächst vorläufige Definitionen der Regel an. Anhand dieser Definitionen motivieren wir dann in den Abschnitten 10.1.2 und 10.1.3 die notwendigen Annotationen des instantiierten Prozedurrumpfs. Da eine erfolgreiche Anwendung der Regeln für tail-rekursive Prozeduren ausgesprochen unwahrscheinlich ist, schließen wir aus Effizienzgründen in Abschnitt 10.1.4 die Anwendung der Regeln für *tail-rekursive* Prozeduren explizit aus. In Abschnitt 10.1.5 geben wir schließlich die endgültigen Definitionen der Regeln an.

10.1.1 Vorläufige Regeldefinitionen

Als vorläufige Definitionen der Regeln „*Unfold structure predicate argument*" und „*Unfold selector argument*" verwenden wir die beiden folgenden Ersetzungsregeln. Die hierbei verwendeten Annotationen haben wir aus den „*Execute*"-Regeln übernommen:

$$\frac{?cons(f^{\,A}(t_1,\ldots,t_n))}{?cons(\sigma_{\xi_{f(\ldots)}}(\mathsf{s}^*_f(\mathsf{un}^{u,s,0}_f(R_f))))},$$

	Label	S-Lim.	U-Lim.
A:	l	s	u

falls $f \in \Sigma \cap \Sigma^{\mathrm{rec}}(P)$, $l \notin C$, $\sigma = \{x_1/\mathsf{a}_{A_0}(t_1),\ldots,x_n/\mathsf{a}_{A_0}(t_n)\}$ und folgendes gilt:

$(?cons(\sigma_{\xi_{f(\ldots)}}(\mathsf{s}^*_f(\mathsf{un}^{u,s,0}_f(R_f))))) \downarrow_{\mathcal{U}}$ enthält keinen rekursiven Prozeduraufruf von f. (*)

(10.2)

$$\frac{sel(f^{\,A}(t_1,\ldots,t_n))}{sel(\sigma_{\xi_{f(\ldots)}}(\mathsf{s}^*_f(\mathsf{un}^{u,s,0}_f(R_f))))},$$

	Label	S-Lim.	U-Lim.
A:	l	s	u

falls $f \in \Sigma \cap \Sigma^{\mathrm{rec}}(P)$, $l \notin C$, $\sigma = \{x_1/\mathsf{a}_{A_0}(t_1),\ldots,x_n/\mathsf{a}_{A_0}(t_n)\}$ und folgendes gilt:

$(sel(\sigma_{\xi_{f(\ldots)}}(\mathsf{s}^*_f(\mathsf{un}^{u,s,0}_f(R_f))))) \downarrow_{\mathcal{U}}$ enthält keinen rekursiven Prozeduraufruf von f. (*)

(10.3)

In den folgenden Abschnitten beschränken wir uns der Einfachheit halber auf eine Diskussion der Regel (10.2). Alle Aussagen gelten dabei in unveränderter Form auch für Regel (10.3)

10.1.2 Annotationen des Prozedurrumpfs

Durch die Anwendungsbedingung $(*)$ der Regel (10.2) ist ein unendlicher Suchraum definiert. Während der symbolischen Auswertung des Terms

$$?cons(\sigma_{\xi_{f(\ldots)}}(\mathsf{s}^*_f(\mathsf{un}^{u,s,0}_f(R_f)))) \tag{10.4}$$

in der Anwendungsbedingung $(*)$ kann die Regel (10.2) erneut für die rekursiven Prozeduraufrufe der Prozedur f überprüft werden. Dadurch entsteht ein endloser rekursiver Abstieg. Um solche endlosen rekursiven Abstiege zu verhindern, enthalten die A-Annotationen das Unfold-Limit. Dieses Unfold-Limit verwenden wir wie folgt: Vor einer Überprüfung der Anwendbarkeit der Regel (10.2) überprüfen wir, ob das Unfold-Limit des Prozeduraufrufs $f(t_1, \ldots, t_n)$ größer als 0 ist. Ist dies nicht der Fall, so wenden wir die Regel nicht an. Ist das Unfold-Limit größer als 0, so überprüfen wir die Anwendungsbedingung $(*)$, wobei wir die Prozeduraufrufe im Prozedurrumpf R_f mit einem entsprechend dekrementierten Unfold-Limit annotieren. Hierbei annotieren wir sowohl rekursive Prozeduraufrufe also auch nicht-rekursive Prozeduraufrufe mit einem dekrementieren Unfold-Limit, obwohl dies zur Vermeidung von endlosen rekursiven Abstiegen, nur für die rekursiven Prozeduraufrufe notwendig ist. Absicht ist dabei den Suchraum zu verkleinern und so eine effiziente Überprüfung der Anwendungsbedingung $(*)$ zu ermöglichen. Hierzu müssen wir jedoch zusätzlich das Unfold-Limit der rekursiven Prozeduraufrufe stärker dekrementieren, als das Unfold-Limit der nicht-rekursiven Prozeduraufrufe. Wir formulieren daher insgesamt die folgende Modifikation:

Modifikation (Unfold-Limit):
Das Unfold-Limit der Prozeduraufrufe im Prozedurrumpf R_f wird durch die Regel (10.2) wie folgt gesetzt: Für nicht-rekursive Prozeduraufrufe wird das Unfold-Limit auf $u-1$ gesetzt und für rekursive Prozeduraufrufe auf $u-2$.

Anmerkung 10.2. Das Unfold-Limit jedes Prozeduraufrufs des Goal-Terms einer initialen Sequenz wird, wie bereits in Kapitel 7 beschrieben, vom ✓eriFun-System auf $u_{\max}$ gesetzt. Für die Version 3.0 des Systems gilt $u_{\max} = 2$. Das bedeutet, dass während Auswertung des instantiierten Prozedurrumpfs durch die Regel (10.2) aufgrund unserer Modifikation die rekursiven Prozeduraufrufe nicht mit Hilfe von „*Unfold*“-Regeln ausgewertet werden dürfen. □

Zur Realisierung der Modifikation definieren wir die folgende Funktion:

$$\begin{aligned}
\mathsf{dec}_f(\texttt{let}^{\,A}\ x\texttt{:=}t\ \texttt{in}\ r\ \texttt{end}) &:= \texttt{let}^{\,A}\ x\texttt{:=}\mathsf{dec}_f(t)\ \texttt{in}\ \mathsf{dec}_f(r)\ \texttt{end}),\\
\mathsf{dec}_f(f^{\langle\ldots,u,\ldots\rangle}(t_1,\ldots,t_n)) &:= f^{\langle\ldots,u-1,\ldots\rangle}(\mathsf{dec}_f(t_1),\ldots,\mathsf{dec}_f(t_n)),\\
\mathsf{dec}_f(g^{\,A}(t_1,\ldots,t_n)) &:= g^{\,A}(\mathsf{dec}_f(t_1),\ldots,\mathsf{dec}_f(t_n)) \text{ falls } f \neq g.
\end{aligned}$$

Wir annotieren dann den Prozedurrumpf R_f in der Regel (10.2) durch den folgenden Ausdruck

$$\mathsf{s}^*_f(\mathsf{dec}_f(\mathsf{un}^{u-1,\ldots,\ldots}_f(R_f))).$$

Für eine effiziente Überprüfung der Anwendungsbedingung $(*)$ sind weitere Beschränkungen der symbolischen Auswertung notwendig:

(1) Quantifikationen und „*Functionality*“-Regeln führen in die Auswertung des Terms (10.4) einen sehr großen Suchraum ein. Um eine effiziente Überprüfung der Anwendungsbedingung $(*)$ zu gewährleisten, müssen wir daher die Verwendung von Quantifikationen und „*Functionality*“-Regeln für $(*)$ verbieten. Hierzu setzen wir die Search-Limits in des Terms (10.4) auf 0.

(2) Die Verwendung der „*Execute*“-Regeln zur Auswertung des Terms (10.4) in $(*)$ ist, wie unsere Fallstudien belegen, nicht notwendig. Wir können daher die Effizienz der symbolischen Auswertung in $(*)$ steigern, ohne die Mächtigkeit des Auswertungskalküls zu beeinträchtigen, in dem wir auf eine Verwendung der „*Execute*“-Regeln für die symbolische Auswertung in $(*)$ verzichten. Das bedeutet, dass Prozeduraufrufe in (10.4) ausschließlich mit Hilfe der „*Unfold*“-Regeln ausgewertet werden. Um die Verwendung der „*Execute*“-Regeln zu verhindern, entfernen wir aus der Menge der anzuwendenden Regel $\mathcal{R}$ alle „*Execute*“-Regeln. Wir definieren hierzu die folgende Funktion:

$$\mathit{no\text{-}execute}(\mathcal{R}) := \mathcal{R} \setminus \{41, \ldots, 52\}\,.$$

Es ergibt sich somit die folgenden Modifikation:

Modifikation (Search-Limit und Regelmenge):
Der Term (10.4) wird in der Anwendungsbedingung $(*)$ ohne die Verwendung von Quantifikationen sowie ohne „*Functionality*“- und „*Execute*“-Regeln ausgewertet. Hierzu wird das Search-Limit auf 0 gesetzt sowie die „*Functionality*“- und „*Execute*“-Regeln aus der Menge der anzuwendenden Regeln entfernt.

Da die Auswertung des Terms (10.4) in der Anwendungsbedingung $(*)$ nun ausgesprochen starken Einschränkungen unterliegt, stellt sich die Frage, ob mit Hilfe von Regel (10.2) überhaupt noch interessante Auswertungen von Prozeduraufrufen möglich sind. Dies ist, wie das folgende Beispiel zeigt, durchaus der Fall:

Beispiel 10.3. Sei P das Programm $\langle D_{\texttt{plus}}, D_{\texttt{times}}\rangle$, wobei $D_{\texttt{plus}}$ die entsprechende Prozedurdefinition aus Abbildung 8.2 und $D_{\texttt{times}}$ die folgende Prozedurdefinition bezeichnen:

$$\begin{array}{rcl} D_{\texttt{times}} & = & \texttt{function times(x : nat, y : nat) : nat} \Leftarrow \\ & & \quad \texttt{if(?0(x),} \\ & & \quad\quad \texttt{0,} \\ & & \quad\quad \texttt{plus(times(pred(x), y), y)).} \end{array}$$

Wir notieren nachfolgend die Terme mit den für das Beispiel relevanten Unfold-Limits. Alle weiteren Komponenten der A-Annotationen werden nicht dargestellt. Mit Hilfe von Regel (10.2) können wir unter Berücksichtigung der formulierten Modifikationen den Prozeduraufruf $\texttt{times}^{\,2}(\texttt{x},\texttt{y})$ des Terms

$$\texttt{if(?0(y), true, ?0(times}^{\,2}\texttt{(x, y)))} \tag{10.5}$$

auswerten und es ergibt sich insgesamt die folgende symbolische Auswertung:

$$\begin{array}{lll} \mathcal{U} \vdash \texttt{if(?0(y), true, }\underline{\texttt{?0(times}^{\,2}\texttt{(x, y))}}\texttt{)} & \overset{(1)}{\rightarrow} \\ \quad \texttt{if(?0(y), true, }\underline{\texttt{?0(if(?0(x), 0, plus}^{\,1}\texttt{(times}^{\,0}\texttt{(x, y), y))))}} & \overset{(2)}{\rightarrow} \\ \quad \texttt{if(?0(y), true, if(?0(x), }\underline{\texttt{?0(0)}}\texttt{, ?0(plus}^{\,1}\texttt{(times}^{\,0}\texttt{(x, y), y))))} & \overset{(3)}{\rightarrow} \end{array}$$

$$\texttt{if}(\texttt{?0}(\texttt{y}),\texttt{true},\texttt{if}(\texttt{?0}(\texttt{x}),\texttt{true},\underline{\texttt{?0}(\texttt{plus}^{\,1}(\texttt{times}^{\,0}(\texttt{x},\texttt{y}),\texttt{y}))})) \xrightarrow{(4)}$$

$$\begin{array}{l}\texttt{if}(\texttt{?0}(\texttt{y}),\\ \quad\texttt{true},\\ \quad\texttt{if}(\texttt{?0}(\texttt{x}),\\ \quad\quad\texttt{true},\\ \quad\quad\underline{\texttt{?0}(\texttt{if}(\texttt{?0}(\texttt{times}^{\,0}(\texttt{x},\texttt{y})),}\\ \quad\quad\quad\underline{\texttt{y},}\\ \quad\quad\quad\underline{\texttt{succ}(\texttt{plus}^{\,0}(\texttt{pred}(\texttt{times}^{\,0}(\texttt{x},\texttt{y}),\texttt{y})))))})\end{array} \xrightarrow{(5)}$$

$$\begin{array}{l}\texttt{if}(\texttt{?0}(\texttt{y}),\\ \quad\texttt{true},\\ \quad\texttt{if}(\texttt{?0}(\texttt{x}),\\ \quad\quad\texttt{true},\\ \quad\quad\texttt{if}(\texttt{?0}(\texttt{times}^{\,0}(\texttt{x},\texttt{y})),\\ \quad\quad\quad\underline{\texttt{?0}(\texttt{y})},\\ \quad\quad\quad\texttt{?0}(\texttt{succ}(\texttt{plus}^{\,0}(\texttt{pred}(\texttt{times}^{\,0}(\texttt{x},\texttt{y}),\texttt{y})))))))\end{array} \xrightarrow{(6)}$$

$$\begin{array}{l}\texttt{if}(\texttt{?0}(\texttt{y}),\\ \quad\texttt{true},\\ \quad\texttt{if}(\texttt{?0}(\texttt{x}),\\ \quad\quad\texttt{true},\\ \quad\quad\texttt{if}(\texttt{?0}(\texttt{times}^{\,0}(\texttt{x},\texttt{y})),\\ \quad\quad\quad\texttt{false},\\ \quad\quad\quad\underline{\texttt{?0}(\texttt{succ}(\texttt{plus}^{\,0}(\texttt{pred}(\texttt{times}^{\,0}(\texttt{x},\texttt{y}),\texttt{y})))))})\end{array} \xrightarrow{(7)}$$

$$\texttt{if}(\texttt{?0}(\texttt{y}),\texttt{true},\texttt{if}(\texttt{?0}(\texttt{x}),\texttt{true},\underline{\texttt{if}(\texttt{?0}(\texttt{times}^{\,0}(\texttt{x},\texttt{y})),\texttt{false},\texttt{false})})) \xrightarrow{(8)}$$

$$\texttt{if}(\texttt{?0}(\texttt{y}),\texttt{true},\underline{\texttt{if}(\texttt{?0}(\texttt{x}),\texttt{true},\texttt{false})}) \xrightarrow{(9)}$$

$$\texttt{if}(\texttt{?0}(\texttt{y}),\texttt{true},\texttt{?0}(\texttt{x}))$$

Es ist zu beachten, dass die Regel (10.2) nicht nur zur Auswertung des Terms $\texttt{?0}(\texttt{times}^{\,2}(\texttt{x},\texttt{y}))$ in Auswertungsschritt (1) verwendet wird, sondern auch zur Auswertung des Terms

$$\texttt{?0}(\texttt{plus}^{\,1}(\texttt{times}^{\,0}(\texttt{x},\texttt{y}),\texttt{y}))$$

in Auswertungsschritt (4). Wir haben somit durch die Regel (10.2) den Term (10.5) auf die Disjunktion $\texttt{if}(\texttt{?0}(\texttt{y}),\texttt{true},\texttt{?0}(\texttt{x}))$ reduzieren können und damit die Prozedur `times` vollständig entfernt. □

10.1.3 Markierung der rekursiven Prozeduraufrufe

Um die rekursiven Aufrufe der Prozedur f im Term

$$?cons(\sigma_{\xi_{f(\ldots)}}(\mathsf{s}^{*}_{f}(\mathsf{un}^{u,0,\cdots}_{f}(R_f)))) \tag{10.6}$$

von Prozeduraufrufen von f zu unterscheiden, die durch die Argumente $t_1,\ldots,t_n$ oder durch das Symbol $*$ in den Term (10.6) eingeführt werden, annotieren wir die rekursiven Prozeduraufrufe mit einer natürlichen Zahl. Hierzu verwenden wir die als Markierung bezeichnete Komponente der A-Annotationen. Um zu überprüfen, ob das Ergebnis der Auswertung des Terms (10.6) keinen rekursiven Aufruf von f mehr enthält, testen wir dann, ob das Ergebnis keinen Prozeduraufruf von f mit der entsprechenden Markierung enthält. Betrachten wir dazu ein Beispiel:

Beispiel 10.4. Sei P das Programm $\langle D_{\texttt{plus}}, D_{\texttt{times}} \rangle$ wobei $D_{\texttt{plus}}$ die Prozedurdefinition aus Abbildung 8.2 bezeichnet und $D_{\texttt{times}}$ die Prozedurdefinition aus Beispiel 10.3. Betrachten wir nun den Term

$$\texttt{?0}(\texttt{plus}(\texttt{x}, \texttt{times}(\texttt{plus}(\texttt{x}, \texttt{y}), \texttt{z}))) \,. \tag{10.7}$$

Für diesen Term ergibt sich für die Überprüfung der Anwendungsbedingung $(*)$ der Regel (10.2) bei Anwendung auf das äußere Vorkommen von `plus` der folgende Term:

$$\texttt{?0}(\texttt{if}(\texttt{?0}(\texttt{x}), \texttt{times}(\texttt{plus}(\texttt{x}, \texttt{y}), \texttt{z}), \texttt{succ}(\texttt{plus}(\texttt{pred}(\texttt{x}), \texttt{times}(\texttt{plus}(\texttt{x}, \texttt{y}), \texttt{z})))))$$

Dieser Term enthält drei Prozeduraufrufe von `plus`, wobei lediglich der Prozeduraufruf

$$\texttt{plus}(\texttt{pred}(\texttt{x}), \texttt{times}(\texttt{plus}(\texttt{x}, \texttt{y}), \texttt{z}))$$

ein rekursiver Prozeduraufruf ist. Um diesen Prozeduraufruf von den anderen `plus`-Aufrufen zu unterscheiden, setzen wir die Markierung des rekursiven Prozeduraufrufs auf 1, wobei wir annehmen, dass die anderen `plus`-Aufrufe mit 0 markiert sind. Es ergibt sich dann die folgende symbolische Auswertung:[2]

$$\begin{aligned}
\mathcal{U} \;\vdash\; & \underline{\texttt{?0}(\texttt{plus}^{\,0}(\texttt{x}, \texttt{times}(\texttt{plus}^{\,0}(\texttt{x}, \texttt{y}), \texttt{z})))} && \rightarrow \\
& \underline{\texttt{?0}(\texttt{if}(\texttt{?0}(\texttt{x}),} \\
& \qquad \underline{\texttt{times}(\texttt{plus}^{\,0}(\texttt{x}, \texttt{y}), \texttt{z}),} \\
& \qquad \underline{\texttt{succ}(\texttt{plus}^{\,1}(\texttt{pred}(\texttt{x}), \texttt{times}(\texttt{plus}^{\,0}(\texttt{x}, \texttt{y}), \texttt{z})))))} && \rightarrow \\
& \texttt{if}(\texttt{?0}(\texttt{x}), \\
& \quad \texttt{?0}(\texttt{times}(\texttt{plus}^{\,0}(\texttt{x}, \texttt{y}), \texttt{z})), \\
& \quad \underline{\texttt{?0}(\texttt{succ}(\texttt{plus}^{\,1}(\texttt{pred}(\texttt{x}), \texttt{times}(\texttt{plus}^{\,0}(\texttt{x}, \texttt{y}), \texttt{z})))))} && \rightarrow \\
& \texttt{if}(\texttt{?0}(\texttt{x}), \texttt{?0}(\texttt{times}(\texttt{plus}^{\,0}(\texttt{x}, \texttt{y}), \texttt{z})), \texttt{false}).
\end{aligned}$$

Da das Ergebnis keinen Prozeduraufruf von `plus` mit der Markierung 1 enthält, wissen wir, dass alle rekursiven Prozeduraufrufe von `plus` entfernt werden konnten und somit eine Anwendung der Regel (10.2) für den Term (10.7) sinnvoll ist. □

Um die eindeutige Markierung zu berechnen, müssen wir alle Markierungen, die bereits für Prozeduraufrufe in den Argumenten $t_1, \ldots, t_n$ vergeben wurden, berücksichtigen. Die Markierung von Prozeduraufrufen für das Symbol $*$ werden durch die Funktion $\mathbf{s}^*_f$ immer mit 0 markiert. Zur Berechnung der eindeutigen Markierung der rekursiven Aufrufe verwenden wir daher die folgende Funktion, wobei wir der Übersichtlichkeit halber lediglich die Markierungen der A-Annotationen darstellen:

$$\mathit{mark}(\sigma) := 1 + \mathit{max}\{m \in \mathbb{N} \mid \exists x \in \mathit{dom}(\sigma)\ \exists t^{\,m} \in \mathcal{ATerm}(P) \text{ mit } \sigma(x) \geq t^{\,m}\}$$

Wir formulieren somit die folgende Modifikation:

Modifikation (Markierung)
Um in der Regel (10.2) die rekursiven Prozeduraufrufe im Term (10.4) eindeutig von den übrigen Prozeduraufrufen von f unterscheiden zu können, werden diese mit der eindeutigen Markierung $\mathit{mark}(\sigma)$ annotiert.

[2] Hierbei notieren wir ausschließlich die für das Beispiel relevanten Markierungen der Terme. Alle weiteren Komponenten der A-Annotationen werden nicht dargestellt.

10.1.4 Ausschluss Tail-Rekursiver Prozeduren

Wir schließen in diesem Abschnitt die Anwendung der Regel (10.2) auf tail-rekursive Prozeduren aus. Wie üblich nennen wir eine Prozedur *tail-rekursiven*, falls rekursive Aufrufe im Rumpf der Prozedur nur als *Ergebnisterm* auftreten, also weder in *Bedingungen* von `case`-Termen, noch als Argument anderer Prozeduraufrufe. Die Termsignatur aller tail-rekursiven Prozeduren des Programms P wird mit $\Sigma^{\text{tail}}(P)$ bezeichnet. Für tail-rekursive Prozeduren erhalten wir während der Auswertung des Terms

$$?cons(\sigma(\mathsf{s}_f^*(\mathsf{dec}_f(\mathsf{un}_f^{u-1,0,m}(R_f)))))$$

ausschließlich rekursive Aufrufe, die wieder in den Termkontext $?cons(\ldots)$ eingebettet sind. Die Situation hat sich also im Vergleich zum ursprünglichen Prozeduraufruf nicht verbessert, da die Einbettung in den Termkontext weitere Auswertungen hier nicht unterstützt. Es ist daher ausgesprochen unwahrscheinlich, dass die rekursiven Aufrufe aufgrund des Termkontexts $?cons(\ldots)$ eliminiert werden können. Wir beschränken daher die Anwendung der Regel (10.2) auf Prozeduren, die nicht tail-rekursiv sind. Es ergibt sich dann, die folgende Modifikation:

Modifikation (nicht-tail-rekursive Prozeduren):
Regel (10.2) wird nicht auf tail-rekursive Prozeduren angewendet.

10.1.5 Vollständige Regeldefinitionen

Die vollständigen Definitionen der Regeln „*Unfold structure predicate argument*" und „*Unfold selector argument*" lauten wie folgt:

63. Unfold structure predicate argument

$$\frac{?cons(f^{\,A}(t_1,\ldots,t_n))}{\mathsf{env}_{\mathcal{U}'}(?cons^{\,A_{\text{std}}}(\sigma_{\xi_{f(\ldots)}}(\mathsf{s}_f^*(\mathsf{dec}_f(\mathsf{un}_f^{u-1,0,m}(R_f))))))},$$

	Label	U-Lim.
A:	l	u

falls $u > 0$, $f \in \Sigma \setminus \Sigma^{\text{tail}}(P)$, $l \notin C$, $\sigma = \{x_1/\mathsf{a}_{A_0}(t_1), \ldots, x_n/\mathsf{a}_{A_0}(t_n)\}$, $m = mark(\sigma)$ und folgendes gilt:

$$\left.\begin{array}{l}(?cons^{\,A_{\text{std}}}(\sigma_{\xi_{f(\ldots)}}(\mathsf{s}_f^*(\mathsf{dec}_f(\mathsf{un}_f^{u-1,0,m}(R_f)))))) \downarrow_{\mathcal{U}'} \text{ enthält} \\ \text{keinen Teilterm der Form } f^{\,\langle\ldots,m,\ldots\rangle}(\ldots).\end{array}\right\}(*)$$

Hierbei bezeichnet $\mathcal{U}'$ die folgende A-Umgebung:

	Reg.
$\mathcal{U}'$:	$no\text{-}execute(\mathcal{R})$

65. Unfold selector argument

$$\frac{sel(f^{\,A}(t_1,\ldots,t_n))}{\mathbf{env}_{\mathcal{U}'}(sel^{\,A_{\mathrm{std}}}(\sigma_{\xi_{f(\ldots)}}(\mathbf{s}^*_f(\mathbf{dec}_f(\mathbf{un}^{u-1,0,m}_f(R_f))))))},$$

	Label	U-Lim.
A:	l	u

falls $u > 0$, $f \in \Sigma \setminus \Sigma^{\mathrm{tail}}(P)$, $l \notin C$, $\sigma = \{x_1/\mathbf{a}_{A_0}(t_1), \ldots, x_n/\mathbf{a}_{A_0}(t_n)\}$, $m = mark(\sigma)$ und folgendes gilt:

$$\left.\begin{array}{l}(sel^{\,A'}(\sigma_{\xi_{f(\ldots)}}(\mathbf{s}^*_f(\mathbf{dec}_f(\mathbf{un}^{u-1,0,m}_f(R_f)))))) \downarrow_{\mathcal{U}'} \text{ enthält keinen Teilterm der Form } f^{\langle\ldots,m,\ldots\rangle}(\ldots).\end{array}\right\}(*)$$

Hierbei bezeichnet $\mathcal{U}'$ die folgende A-Umgebung:

	Reg.
$\mathcal{U}'$:	$no\text{-}execute(\mathcal{R})$

Die Ergebnisterme der Regeln werden durch die Funktion $\mathbf{env}'_{\mathcal{U}}$ mit der A-Umgebung $\mathcal{U}'$ annotiert. Diese Annotierung stellt sicher, dass die Ergebnisterme mit der gleichen A-Umgebung ausgewertet werden, wie die Terme in der entsprechenden Anwendungsbedingung $(*)$. Wir vermeiden dadurch, dass trotz einer erfolgreichen Überprüfung der Anwendungsbedingung $(*)$ in der eigentlichen symbolischen Auswertung die rekursiven Prozeduraufrufe nicht eliminiert werden können (siehe Beispiel F.7 in Anhang F). Weiter stellen wir dadurch sicher, dass die Ergebnisterme genauso effizient ausgewertet werden, wie die Terme in den entsprechenden Anwendungsbedingungen $(*)$.

10.2 Regeln für Gleichungen und case-Terme

Wir definieren nun die Auswertungsregeln für die Termkontexte der Form

$$\begin{array}{c} l{=}\ldots, \\ \ldots{=}r, \\ \mathtt{case}(\ldots, cons^* : t^*)\,. \end{array} \tag{10.8}$$

Für diese Termkontexte kann es durchaus sinnvoll sein einen Prozeduraufruf einer tail-rekursiven Prozedur auszuwerten, da die in den Termkontexten zusätzlich enthaltenen Terme l, r bzw. t^* eine Elimination der rekursiven Prozeduraufrufe begünstigen. Für die Prozedurdefinition $D_{\mathtt{plus'}}$ aus Beispiel 9.23 und den Term

$$\mathtt{plus'(pred(x),succ(y))=plus'(x,y)}$$

können wir beispielsweise die Prozeduraufrufe der rechten Seite der Gleichung eliminieren, in dem wir den Prozeduraufruf auswerten:

$$\begin{array}{lll} \mathcal{U} \vdash & \underline{\mathtt{plus'(pred(x),succ(y))=if(?0(x),y,plus'(pred(x),succ(y)))}} & \rightarrow \\ & \mathtt{if(?0(x),} & \\ & \quad \mathtt{plus'(pred(x),succ(y))=y,} & \rightarrow \\ & \quad \underline{\mathtt{plus'(pred(x),succ(y))=plus'(pred(x),succ(y)))}} & \\ & \mathtt{if(?0(x),plus'(pred(x),succ(y))=y,true).} & \end{array}$$

Der Grund für die erfolgreiche Elimination ist, dass die linke Seite der Gleichung mit dem rekursiven Aufruf übereinstimmt, der durch die Auswertung der rechten Seite der Gleichung entsteht. Dieser rekursive Aufruf kann daher im zweiten Auswertungsschritt entfernt werden. Es ergibt sich daher der folgende Unterschied zu den Regeldefinitionen in Abschnitt 10.1:

Unterschied zu den Regeldefinitionen in Abschnitt 10.1:
Die Auswertungsregeln für die Termkontexte (10.8) berücksichtigen, im Gegensatz zu den Regeln für die Termkontexte $?cons(\ldots)$ und $sel(\ldots)$, tailrekursive Prozeduren.

Weiter ist für die Definitionen der Auswertungsregeln für die Termkontexte (10.8) zu beachten, dass aus Effizienzgründen die Unfold-Limits und die Search-Limits in den Termen l, r bzw. t^* aus den Kontexten auf 0 gesetzt werden. Die Regeldefinitionen lauten dann wie folgt:

67. Unfold case condition

$$\frac{\mathsf{case}(f^{\,A}(t_1,\ldots,t_n), cons^* : t^*)}{\mathsf{env}_{\mathcal{U}'}(\mathsf{case}^{\,A_{\mathrm{std}}}(\sigma_{\xi_{f(\ldots)}}(\mathsf{s}^*_f(\mathsf{dec}_f(\mathsf{un}^{u-1,0,m}_f(R_f)))), cons^* : \mathsf{a}_{A_0}(t^*)))},$$

falls $u > 0$, $f \in \Sigma \cap \Sigma^{\mathrm{rec}}(P)$, $l \notin C$, $\sigma = \{x_1/\mathsf{a}_{A_0}(t_1), \ldots, x_n/\mathsf{a}_{A_0}(t_n)\}$, $m = mark(\sigma)$ und folgendes gilt:

$$\left.\begin{array}{l}(\mathsf{case}^{\,A_{\mathrm{std}}}(\sigma_{\xi_{f(\ldots)}}(\mathsf{s}^*_f(\mathsf{dec}_f(\mathsf{un}^{u-1,0,m}_f(R_f)))), cons^* : \mathsf{a}_{A_0}(t^*)))\downarrow_{\mathcal{U}'} \\ \text{enthält keinen Teilterm der Form } f^{\,\langle\ldots,m,\ldots\rangle}(\ldots).\end{array}\right\}(*)$$

Hierbei bezeichnet $\mathcal{U}'$ die folgende A-Umgebung und A die folgende A-Annotation:

	Reg.
$\mathcal{U}'$:	$no\text{-}execute(\mathcal{R})$

	Lab.	U-Lim.
A:	l	u

69. Unfold left equality argument

$$\frac{f^{\,A}(t_1,\ldots,t_n)\text{=}r}{\mathsf{env}_{\mathcal{U}'}(\sigma_{\xi_{f(\ldots)}}(\mathsf{s}^*_f(\mathsf{dec}_f(\mathsf{un}^{u-1,0,m}_f(R_f))))\text{=}^{\,A_{\mathrm{std}}}\mathsf{a}_{A_0}(r))},$$

	Label	U-Lim.
A:	l	u

falls $u > 0$, $f \in \Sigma \cap \Sigma^{\mathrm{rec}}(P)$, $l \notin C$, $\sigma = \{x_1/\mathsf{a}_{A_0}(t_1), \ldots, x_n/\mathsf{a}_{A_0}(t_n)\}$, $m = mark(\sigma)$ und folgendes gilt:

$$\left.\begin{array}{l}(\sigma_{\xi_{f(\ldots)}}((\mathsf{s}^*_f(\mathsf{dec}_f(\mathsf{un}^{u-1,0,m}_f(R_f))))\text{=}^{\,A_{\mathrm{std}}}\mathsf{a}_{A_0}(r))\downarrow_{\mathcal{U}'} \text{ enthält} \\ \text{keinen Teilterm der Form } f^{\,\langle\ldots,m,\ldots\rangle}(\ldots).\end{array}\right\}(*)$$

Hierbei bezeichnet $\mathcal{U}'$ die folgende A-Umgebung:

	Reg.
$\mathcal{U}'$:	$no\text{-}execute(\mathcal{R})$

71. Unfold right equality argument

$$\frac{l{=}f^{\,A}(t_1,\ldots,t_n)}{\mathbf{env}_{\mathcal{U}'}(\mathbf{a}_{A_0}(l){=}^{\,A_{\mathrm{std}}}\sigma_{\xi_{f(\ldots)}}(\mathbf{s}^*_f(\mathbf{dec}_f(\mathbf{un}_f^{u-1,0,m}(R_f)))))},$$

	Label	U-Lim.
A:	l	u

falls $u > 0$, $f \in \Sigma \cap \Sigma^{\mathrm{rec}}(P)$, $l \notin C$, $\sigma = \{x_1/\mathbf{a}_{A_0}(t_1), \ldots, x_n/\mathbf{a}_{A_0}(t_n)\}$, $m = mark(\sigma)$ und folgendes gilt:

$$\left.\begin{array}{l}(\mathbf{a}_{A_0}(l){=}^{\,A_{\mathrm{std}}}\sigma_{\xi_{f(\ldots)}}(\mathbf{s}^*_f(\mathbf{dec}_f(\mathbf{un}_f^{u-1,0,m}(R_f))))) \downarrow_{\mathcal{U}'} \text{ enthält} \\ \text{keinen Teilterm der Form } f^{\,\langle\ldots,m,\ldots\rangle}(\ldots).\end{array}\right\}(*)$$

Hierbei bezeichnet $\mathcal{U}'$ die folgende A-Umgebung:

	Reg.
$\mathcal{U}'$:	$no\text{-}execute(\mathcal{R})$

10.3 Regeln für zusammengesetzte Termkontexte

Wir haben für alle in (10.1) aufgezählten Termkontexte Auswertungsregeln definiert und ermöglichen dadurch die Auswertung von Prozeduraufrufen innerhalb dieser Termkontexte. Eine Auswertung aufgrund eines „zusammengesetzten" Termkontexts wie `?0(pred(...))` ist jedoch mit Hilfe dieser Regeln nicht möglich. Betrachten wir dazu ein Beispiel:

Beispiel 10.5. Sei P das Programm $\langle D_{\mathtt{plus}}\rangle$ wobei $D_{\mathtt{plus}}$ die Prozedurdefinition aus Abschnitt 8.2 bezeichnet. Der Term `?0(pred(plus(x,y)))` ist mit unseren aktuell definierten Regeln nicht auswertbar. Insbesondere kann für den Teilterm `pred(plus(x,y))` die Regel „*Unfold selector argument*" nicht angewendet werden, da wir durch die Auswertung des Prozeduraufrufs den Term

$$\mathtt{if(?0(x),pred(y),plus(pred(x),y))}$$

erhalten und damit offensichtlich den rekursiven Prozeduraufruf von `plus` nicht entfernen konnten. Berücksichtigt man jedoch während der Auswertung des instantiierten Prozedurrumpfs den gesamten Termkontext `?0(pred(...))` so ergibt sich die folgende symbolische Auswertung:

$$\begin{array}{lll}
\mathcal{U} \vdash & \mathtt{?0(}\underline{\mathtt{pred(if(?0(x),y,succ(plus(pred(x),y)))))}\mathtt{)} & \overset{(1)}{\rightarrow} \\
& \mathtt{?0(if(?0(x),pred(y),}\underline{\mathtt{pred(succ(plus(pred(x),y)))))}\mathtt{)} & \overset{(2)}{\rightarrow} \\
& \underline{\mathtt{?0(if(?0(x),pred(y),plus(pred(x),y)))}} & \overset{(3)}{\rightarrow} \\
& \mathtt{if(?0(x),?0(pred(y)),}\underline{\mathtt{?0(plus(pred(x),y))}}\mathtt{))} & \overset{(4)}{\rightarrow} \\
& \mathtt{if(?0(x),} & \\
& \quad \mathtt{?0(pred(y)),} & \\
& \quad \underline{\mathtt{?0(if(?0(pred(x)),}} & \overset{(5)}{\rightarrow} \\
& \qquad \underline{\mathtt{y,}} & \\
& \qquad \underline{\mathtt{succ(plus(pred(pred(x)),y)))))}} &
\end{array}$$

```
if(?0(x),
   ?0(pred(y)),
   if(?0(pred(x)),                                   (6)
      ?0(y),                                         →
      ?0(succ(plus(pred(pred(x)),y))))))

if(?0(x),?0(pred(y)),if(?0(pred(x)),?0(y),false))
```

Unter Berücksichtigung des vollständigen Termkontexts können wir also den rekursiven Prozeduraufruf von `plus` eliminieren und eine Auswertung des Prozeduraufrufs ist somit sinnvoll. □

Um symbolische Auswertungen für die typischen zusammengesetzten Termkontexte $?cons(sel(\ldots))$, $sel(\ldots){=}r$ und $l{=}sel(\ldots)$ zu ermöglichen, definieren wir die Auswertungsregeln „*Unfold structure predicate selector argument*“, „*Unfold left equality selector argument*“ und „*Unfold right equality selector argument*“. Hierbei ist der folgende Unterschied zu den Regeldefinitionen aus Abschnitt 10.1 zu beachten:

Unterschied zu den Regeldefinitionen in Abschnitt 10.1:
Das Unfold-Limit der rekursiven Prozeduraufrufe im instantiierten Rumpf wird auf $u-1$ gesetzt.

Dieser Unterschied ist notwendig, da, wie Auswertungsschritt (4) der symbolischen Auswertung in Beispiel 10.5 belegt, für eine erfolgreiche Auswertung des instantiierten Prozedurrumpfs unter Berücksichtigung der zusammengesetzten Termkontexte eine entsprechende „*Unfold*“-Regel auf die rekursiven Prozeduraufrufe angewendet werden muss. Weiter ist zu beachten, dass die Regeln wieder nur für nicht-tail-rekursive Prozeduren angewendet werden.

73. Unfold structure predicate selector argument

$$\frac{?cons(sel(f^{\,A}(t_1,\ldots,t_n)))}{\mathbf{env}_{\mathcal{U}'}(?cons^{\,A_{\mathrm{std}}}(sel^{\,A_{\mathrm{std}}}(\sigma_{\xi_{f(\ldots)}}(\mathbf{s}^*_f(\mathbf{un}^{u-1,0,m}_f(R_f))))))},$$

	Label	U-Lim.
A:	l	u

falls $u > 0$, $f \in \Sigma \setminus \Sigma^{\mathrm{tail}}(P)$, $l \notin C$, $\sigma = \{x_1/\mathbf{a}_{A_0}(t_1),\ldots,x_n/\mathbf{a}_{A_0}(t_n)\}$, $m = mark(\sigma)$ und folgendes gilt:

$$\left.\begin{array}{l}(?cons^{\,A_{\mathrm{std}}}(sel^{\,A_{\mathrm{std}}}(\sigma_{\xi_{f(\ldots)}}(\mathbf{s}^*_f(\mathbf{un}^{u-1,0,m}_f(R_f)))))) \downarrow_{\mathcal{U}'} \text{ enthält keinen Teilterm der Form } f^{\,\langle\ldots,m,\ldots\rangle}(\ldots).\end{array}\right\}(*)$$

Hierbei bezeichnet $\mathcal{U}'$ die folgende A-Umgebung:

	Reg.
$\mathcal{U}'$:	$no\text{-}execute(\mathcal{R})$

75. Unfold left equality selector argument

$$\frac{sel(f^{\,A}(t_1,\ldots,t_n))=r}{\mathbf{env}_{\mathcal{U}'}(sel^{\,A_{\mathrm{std}}}(\sigma_{\xi_{f(\ldots)}}(\mathbf{s}^*_f(\mathbf{un}_f^{u-1,0,m}(R_f))))=^{A_{\mathrm{std}}}\mathbf{a}_{A_0}(r))},$$

	Label	U-Lim.
A:	l	u

falls $u > 0$, $f \in \Sigma \setminus \Sigma^{\mathrm{tail}}(P)$, $l \notin C$, $\sigma = \{x_1/\mathbf{a}_{A_0}(t_1), \ldots, x_n/\mathbf{a}_{A_0}(t_n)\}$, $m = mark(\sigma)$ und folgendes gilt:

$$\left.\begin{array}{l}(sel^{\,A_{\mathrm{std}}}(\sigma_{\xi_{f(\ldots)}}(\mathbf{s}^*_f(\mathbf{un}_f^{u-1,0,m}(R_f))))=^{A_{\mathrm{std}}}\mathbf{a}_{A_0}(r))\downarrow_{\mathcal{U}'} \text{ enthält keinen Teilterm der Form } f^{\langle\ldots,m,\ldots\rangle}(\ldots).\end{array}\right\}(*)$$

Hierbei bezeichnet $\mathcal{U}'$ die folgende A-Umgebung:

	Reg.
$\mathcal{U}'$:	$no\text{-}execute(\mathcal{R})$

77. Unfold right equality selector argument

$$\frac{l=sel(f^{\,A}(t_1,\ldots,t_n))}{\mathbf{env}_{\mathcal{U}'}(\mathbf{a}_{A_0}(l)=^{A_{\mathrm{std}}}sel^{\,A_{\mathrm{std}}}(\sigma_{\xi_{f(\ldots)}}(\mathbf{s}^*_f(\mathbf{un}_f^{u-1,0,m}(R_f)))))},$$

	Label	U-Lim.
A:	l	u

falls $u > 0$, $f \in \Sigma \setminus \Sigma^{\mathrm{tail}}(P)$, $l \notin C$, $\sigma = \{x_1/\mathbf{a}_{A_0}(t_1), \ldots, x_n/\mathbf{a}_{A_0}(t_n)\}$, $m = mark(\sigma)$ und folgendes gilt:

$$\left.\begin{array}{l}(\mathbf{a}_{A_0}(l)=^{A_{\mathrm{std}}}sel^{\,A_{\mathrm{std}}}(\sigma_{\xi_{f(\ldots)}}(\mathbf{s}^*_f(\mathbf{un}_f^{u-1,0,m}(R_f)))))\downarrow_{\mathcal{U}'} \text{ enthält keinen Teilterm der Form } f^{\langle\ldots,m,\ldots\rangle}(\ldots).\end{array}\right\}(*)$$

Hierbei bezeichnet $\mathcal{U}'$ die folgende A-Umgebung:

	Reg.
$\mathcal{U}'$:	$no\text{-}execute(\mathcal{R})$

10.4 Regel für einfache Prozeduraufrufe

Die bisher definierten „*Unfold*“-Regeln ermöglichen nur die Auswertung von Prozeduraufrufen, die in einem der Termkontexte aus (10.1) enthalten sind. So können wir beispielsweise für die in Abbildung 10.1 definierte Prozedur `member` den Prozeduraufruf von `member` im `then`-Teil des Terms

$$\texttt{if}(\texttt{member}(\texttt{n},\texttt{tl}(\texttt{l})),\texttt{member}(\texttt{n},\texttt{l}),\texttt{true})$$

mit Hilfe der „*Unfold*“-Regeln nicht auswerten. Ersetzen wir jedoch den Prozeduraufruf durch den entsprechend instantiierten Prozedurrumpf, so ergibt sich die folgende Auswertung:

$$
\begin{array}{rcl}
D_{\texttt{member}} & = & \texttt{function member(n : @a, l : list[@a]) : bool} \Leftarrow \\
& & \quad \texttt{if(?empty(t),} \\
& & \qquad \texttt{false,} \\
& & \qquad \texttt{if(n=hd(l),} \\
& & \qquad\quad \texttt{true,} \\
& & \qquad\quad \texttt{member(n, tl(l))))}
\end{array}
$$

Abbildung 10.1: Prozedurdefinition von `member`.

$$
\begin{array}{rcll}
 & & \texttt{if(member(n, tl(l)),} & \\
\mathcal{U} & \vdash & \quad \texttt{if(?empty(l), false, if(n=hd(l), true, } \underline{\texttt{member(n, tl(l))}}\texttt{)),} & \rightarrow \\
 & & \quad \texttt{true)} & \\
 & & \texttt{if(member(n, tl(l)),} & \\
 & & \quad \texttt{if(?empty(l), false, } \underline{\texttt{if(n=hd(l), true, true)}}\texttt{),} & \rightarrow \\
 & & \quad \texttt{true)} & \\
 & & \texttt{if(member(n, tl(l)), } \neg\texttt{?empty(l), true)} &
\end{array}
$$

Das bedeutet, dass durch Auswertung des instantiierten Prozedurrumpfs der rekursive Prozeduraufruf von `member` entfernt werden kann und somit eine Auswertung des Prozeduraufrufs `member(n, l)` sinnvoll ist. Der Grund für die erfolgreiche Elimination des rekursiven Aufrufs ist, dass in der Hypothesenmenge, unter der der Prozeduraufruf ausgewertet wird, ein entsprechender Prozeduraufruf von `member` enthalten ist. Allgemein werten wir daher einen Prozeduraufruf $\sigma(f(\ldots))$ unabhängig vom Termkontext aus, falls in der globalen oder lokalen Hypothesenmenge ein Prozeduraufruf von f enthalten ist und durch die Auswertung des instantiierten Prozedurrumpfs $\sigma_{f(\ldots)}(R_f^*)$ die rekursiven Aufrufe von f eliminiert werden können:

53. Unfold procedure call

$$
\frac{f^{\,A}(t_1, \ldots, t_n)}{\mathbf{env}_{\mathcal{U}'}(\sigma_{\xi_{f(\ldots)}}(\mathbf{s}_f^*(\mathbf{dec}_f(\mathbf{un}_f^{u-1,0,m}(R_f)))))}, \qquad \begin{array}{|l|l|l|} \hline & \text{Label} & \text{U-Lim.} \\ \hline A: & l & u \\ \hline \end{array}
$$

falls $u > 0$, $f \in \Sigma \cap \Sigma^{\mathrm{rec}}(P)$, $l \notin C$, $\sigma = \{x_1/\mathbf{a}_{A_0}(t_1), \ldots, x_n/\mathbf{a}_{A_0}(t_n)\}$, $m = mark(\sigma)$ und $f \in \bigcup_{h \in H_{\mathrm{L}} \cup H_{\mathrm{G}}} proc_P(h)$. Weiter muss folgendes gelten:

$$
\left.\begin{array}{l} (\sigma_{\xi_{f(\ldots)}}(\mathbf{s}_f^*(\mathbf{dec}_f(\mathbf{un}_f^{u-1,0,m}(R_f))))) \downarrow_{\mathcal{U}'} \text{ enthält keinen Teil-} \\ \text{term der Form } f^{\,\langle \ldots, m, \ldots \rangle}(\ldots). \end{array}\right\} (*)
$$

Hierbei bezeichnet $\mathcal{U}'$ die folgende A-Umgebung:

	Reg.
$\mathcal{U}'$:	$no\text{-}execute(\mathcal{R})$

10.5 Anmerkungen zur Implementierung

Wie für die „*Execute*“-Regeln gilt auch für die „*Unfold*“-Regeln, dass für kommutative Prozeduren die Anwendung einer „*Unfold*“-Regel unter Umständen erst nach Vertauschen der Argumente möglich ist. Wir definieren daher in Anhang D für jede „*Unfold*“-Regel eine entsprechend kommutierte Version. Wie bei den „*Execute*“-Regeln weicht die Implementierung der kommutativen Versionen der „*Unfold*“-Regeln im ✓`eriFun`-System insofern von unserer Definition ab, dass die Argumente der rekursiven Funktionsaufrufe im instantiierten Prozedurrumpf zusätzlich vertauscht

werden. Diese zusätzliche Vertauschung hat wieder lediglich „optische“ Gründe und hat für die weitere Auswertung und Verwendung der Terme keine Konsequenzen.

Die Implementierung der Regeln für zusammengesetzte Termkontexte im ✓`eriFun`-System weicht von unserer Definition der Regeln in Abschnitt 10.3 ab. In der Implementierung werden diese Regeln nur für Prozeduren angewendet, die keine rekursiven Aufrufe in den Bedingungen der `case`-Terme des Prozedurrumpfs enthalten. Diese zusätzliche Einschränkung beschleunigt die symbolische Auswertung in einigen Fallstudien deutlich. Der Grund hierfür ist, dass in diesen Fallstudien einige sehr umfangreiche Prozeduren vorhanden sind, die zufälligerweise auch einen rekursiven Prozeduraufruf in einer Bedingung enthalten. Die Prozeduraufrufe dieser Prozeduren werden dann von den Regeln für zusammengesetzte Termkontexte nicht behandelt. Da jedoch der Aufwand für die Überprüfung der Anwendungsbedingungen der Regeln nicht in Zusammenhang mit den rekursiven Aufrufen in den Bedingungen der `case`-Terme steht, ist es keine gute Idee, die Komplexität einer Prozedur anhand dieser rekursiven Aufrufe zu messen. Besser wäre, die Regeln generell auf Prozeduren zu beschränken, deren Rumpf eine gewisse Größe nicht überschreitet. Da die Beschränkung auf Prozeduren, die keine rekursiven Aufrufe in den Bedingungen der `case`-Terme enthalten, nur als „ad-hoc“ Maßnahme angesehen werden kann, haben wir diese Beschränkung nicht in unsere Regeldefinitionen aufgenommen.

Kapitel 11

Die „*Assumption*“-Regeln

Um die Gültigkeit einer Sequenz $H, IH \vdash g$ durch symbolische Auswertung des Goal-Terms g zu beweisen, ist es im Allgemeinen notwendig die vom ✓eriFun-System bereits bewiesenen Lemmata und die Induktionshypothesen der Sequenz zu verwenden. Hierzu gehen wir wie folgt vor:

Vorgehensweise (Verwendung von Quantifikationen):
Die Klauseln $C_i = \{l^i_1, \ldots, l^i_{n_i}\}$ der mit den Lemmata und Induktionshypothesen assoziierten Klauselmengen $\{C_1, \ldots, C_n\}$ können als Implikationen der Form

$$\overline{l^i_1} \wedge \ldots \wedge \overline{l^i_{j-1}} \wedge \overline{l^i_{j+1}} \wedge \ldots \wedge \overline{l^i_{n_i}} \Rightarrow l^i_j$$

aufgefasst werden. Um die Gültigkeit eines Atoms a mit $a \simeq_{\mathcal{U}} \sigma_\xi(l^i_j)$ (bzw. die Ungültigkeit eines Atoms a mit $a \simeq_{\mathcal{U}} \overline{\sigma_\xi(l^i_j)}$) zu zeigen, muss dann die Ungültigkeit der Literale $\sigma_\xi(l^i_k)$ mit $k \in \{1, \ldots, n\} \setminus \{j\}$ nachgewiesen werden. Hierzu werden die Literale symbolisch ausgewertet. Können alle Literale $\sigma_\xi(l^i_k)$ zu `false` ausgewertet werden, so kann das Atom a durch `true` (bzw. `false`) ersetzt werden.

Das Literal $l^i_j \in C_i$ bezeichnen wir nachfolgend als *Pattern-Literal* und die Literale $\sigma_\xi(l^i_k)$ mit $k \in \{1, \ldots, n\} \setminus \{j\}$ als *Subgoals*. Das Atom a bezeichnen wir als *Proof-Goal*. Zur Realisierung unserer Vorgehensweise definieren wir in diesem Kapitel die folgenden Auswertungsregeln:

⋮ ⋮
61. *Affirmative assumption*
62. *Negative assumption*
⋮ ⋮

Diese Auswertungsregeln sind - neben den „*Execute*“- und „*Unfold*“-Regeln - die zentralen Regeln des Auswertungskalküls. Um eine möglichst effiziente und mächtige Verwendung von Quantifikationen durch diese Regeln zu gewährleisten, müssen wir die Subgoals entsprechend effizient auswerten sowie die für das Proof-Goal relevanten Klauseln und Pattern-Literale in den Regeln betrachten. Es sind somit die beiden folgenden Fragen zu klären:

(1) Wie werden die Subgoals symbolisch ausgewertet?

(2) Wie werden die Klauseln und Pattern-Literale für ein Proof-Goal bestimmt?

Zur Definition der Regeln gehen wir daher wie folgt vor. Wir führen zunächst in Abschnitt 11.1 vorläufige Regeldefinitionen ein, um die Betrachtungen in den nachfolgenden Abschnitten zu vereinfachen. Anschließend beschäftigen wir uns in Abschnitt 11.2 mit der effizienten Auswertung der Subgoals. Die Ergebnisse dieses Abschnitts fügen wir dann in Abschnitt 11.3 in die vorläufigen Regeldefinitionen ein. Danach gehen wir in Abschnitt 11.4 auf die Frage ein, wie die Klauseln und die Pattern-Literale für ein Proof-Goal bestimmt werden. Basierend auf den Antworten geben wir dann in Abschnitt 11.5 die vollständigen Definitionen der Regeln an.

11.1 Vorläufige Regeldefinitionen

Für die Definition der vorläufigen Regeln sind die folgenden Punkte zu beachten:

- Bei den Regeln handelt es sich um rekursive Auswertungsregeln. Wir definieren daher die Regeln entsprechend dem in Abschnitt 7.7 angegebenen Entwurfsprinzips zur Transparenz rekursiver Auswertungsregeln. Das bedeutet, wir definieren keine Ersetzungsregeln der Form

  $$\frac{a}{\texttt{true}} \qquad \text{bzw.} \qquad \frac{a}{\texttt{false}}$$

  sondern Ersetzungsregeln der Form[1]

  $$\frac{a}{NOR(\sigma_\xi(C) - \sigma_\xi(lit))} \qquad \text{bzw.} \qquad \frac{a}{OR(\sigma_\xi(C) - \sigma_\xi(lit))}.$$

  Durch diese Ersetzungsregeln ist sichergestellt, dass die symbolischen Auswertungen der Subgoals in der eigentlichen symbolischen Auswertung nachvollzogen werden können. Es ist zu beachten, dass $NOR(\sigma_\xi(C) - \sigma_\xi(lit))$ und $OR(\sigma_\xi(C) - \sigma_\xi(lit))$ genau dann zu `true` bzw. `false` ausgewertet werden können, wenn jedes Subgoal in $\sigma_\xi(C) - \sigma_\xi(lit)$ zu `false` ausgewertet werden kann.

- Ist das Proof-Goal ein Strukturtest, dann ist es sinnvoll dieses Proof-Goal nicht nur als Strukturtest zu betrachten, sondern auch als Strukturgleichung. Wir definieren hierzu die folgende Funktion:

  $$\mathit{struct\text{-}set}_P(a) := \begin{cases} \{a, \mathit{struct\text{-}eq}_P(a)\} & \text{falls } a = \texttt{?}\mathit{cons}_i(t), \\ \{a\} & \text{sonst} \end{cases}$$

  mit

  $$\mathit{struct\text{-}eq}_P(\texttt{?}\mathit{cons}_i(t)) := t\texttt{=}\mathit{cons}_i(\mathit{sel}_{i,1}(t), \ldots, \mathit{sel}_{i,n_i}(t)).$$

  Die Funktion $\mathit{struct\text{-}set}_P$ liefert damit für ein Proof-Goal a alle Varianten des Proof-Goals, die sinnvoller Weise bei der Anwendung einer Quantifikation betrachtet werden sollten.

Bevor wir nun die vorläufigen Regeldefinitionen angeben, führen wir den Begriff des $\approx$-*Matchers* ein, um die Regeln kompakt repräsentieren zu können.

Definition 11.1 ($\approx$-Matcher)**.** Sei P ein Programm und $s, t \in \mathcal{Term}(P)$. Sei weiter $\approx$ eine Äquivalenzrelationen auf Termen. Ein Paar σ_ξ mit $\xi \in \mathcal{Subst}_{tv'(s)}(\Omega(P))$

[1] $\sigma_\xi(C_i) - \sigma_\xi(l_i^j)$ ist eine Kurzschreibweise für $\sigma_\xi(C_i) \setminus_{\simeq_\mathcal{U}} \{\sigma_\xi(l_j^i)\}$. $NOR(C)$ bezeichnet die Konjunktion der Komplemente der Literale in C.

und $\sigma \in \mathcal{Subst}_{fv'(\xi(s))}(P)$ heißt $\approx$-*Matcher von* s *für* t falls $\sigma_\xi(s) \approx t$ gilt. Die Erweiterung der Äquivalenzrelation $\approx$ auf Matcher ist wie folgt definiert:

$$\sigma_\xi \approx \sigma'_{\xi'} \quad \text{gdw.} \quad \begin{array}{l} \xi = \xi',\ dom(\sigma) = dom(\sigma') \text{ und für} \\ \text{alle } x \in dom(\sigma) \text{ gilt } \sigma(x) \approx \sigma'(x). \end{array}$$

Die Menge $match_\approx(s,t)$ aller $\approx$-Matcher von s für t ist dann gegeben durch

$$match_\approx(s,t) := \{\varepsilon_\approx(\sigma_\xi) \mid \sigma_\xi \text{ ist ein } \approx\text{-Matcher von } s \text{ für } t\}.^2$$

□

Es ist zu beachten, dass in Abhängigkeit der Relation $\approx$ mehrere Matcher für Terme s und t existieren können. Die Beschränkung der Menge $match_\approx(s,t)$ auf die kanonischen Repräsentanten der Matcher ist notwendig, um die Menge $match_\approx(s,t)$ nicht mit „$\approx$-Dubletten“ aufzublähen.

Beispiel 11.2. Wir nehmen an, dass die Kommutativität der Prozedur `plus` bekannt ist. Dann sind die Paare σ^1_ϵ und σ^2_ϵ mit

$$\begin{array}{lcl} \sigma^1 & = & \{\mathtt{x/plus(u,v)}\} \\ \sigma^2 & = & \{\mathtt{x/plus(v,u)}\} \end{array}$$

die beiden einzigen $\simeq_\mathcal{U}$-Matcher von `x=0` für `plus(u,v)=0`. Da $\sigma^1_\epsilon \simeq_\mathcal{U} \sigma^2_\epsilon$ gilt, besteht die Menge

$$match_{\simeq_\mathcal{U}}(\mathtt{x=0}, \mathtt{plus(u,v)=0})$$

aus genau einem der beiden Matcher. □

Unsere vorläufigen Regeldefinitionen sehen dann wie folgt aus:

$$\left.\begin{array}{l} \dfrac{a}{NOR(\sigma_\xi(C) - \sigma_\xi(lit))}, \\ \\ \text{falls } \mathtt{e}(a) \in \mathcal{At}(P) \text{ gilt und ein } C \in \bigcup_{q\in\mathcal{Q}} \mathcal{C}_q \text{ sowie ein } lit \in C \text{ existieren, so dass für eine } a' \in \textit{struct-set}_P(a) \text{ und ein } \sigma_\xi \in match_{\simeq_\mathcal{U}}(lit, a') \\ \text{die folgenden Bedingungen erfüllt sind:} \\ \quad (1)\ tv'(C) \subseteq tv'(lit) \text{ und } fv'(\xi(C)) \subseteq fv'(\xi(lit)). \\ \quad (2)\ \forall d \in (\sigma_\xi(C) - \sigma_\xi(lit)) \text{ gilt } \mathcal{U} \vdash d \to^! \mathtt{false}. \end{array}\right\} \quad (11.1)$$

$$\left.\begin{array}{l} \dfrac{a}{OR(\sigma_\xi(C) - \sigma_\xi(lit))}, \\ \\ \text{falls } \mathtt{e}(a) \in \mathcal{At}(P) \text{ gilt und ein } C \in \bigcup_{q\in\mathcal{Q}} \mathcal{C}_q \text{ sowie ein } lit \in C \text{ existieren, so dass für eine } a' \in \textit{struct-set}_P(a) \text{ und ein } \sigma_\xi \in match_{\simeq_\mathcal{U}}(\overline{lit}, a') \\ \text{die folgenden Bedingungen erfüllt sind:} \\ \quad (1)\ tv'(C) \subseteq tv'(lit) \text{ und } fv'(\xi(C)) \subseteq fv'(\xi(lit)). \\ \quad (2)\ \forall d \in (\sigma_\xi(C) - \sigma_\xi(lit)) \text{ gilt } \mathcal{U} \vdash d \to^! \mathtt{false}. \end{array}\right\} \quad (11.2)$$

Die Lemmata und Induktionshypothesen werden während der symbolischen Auswertung durch die in der Multimenge $\mathcal{Q}$ der A-Umgebung enthaltenen Quantifikationen repräsentiert. Der Ausdruck $C \in \bigcup_{q\in\mathcal{Q}} \mathcal{C}_q$ wählt somit eine Klausel eines

[2] Mit $\varepsilon_\approx(\sigma_\xi)$ bezeichnen wir den kanonischen Repräsentanten der Äquivalenzklasse $[\sigma_\xi]_\approx$ (vergleiche Anhang A).

Lemmas bzw. einer Induktionshypothese aus. Die Anwendungsbedingung (1) stellt sicher, dass alle freien Typ- und Termvariablen der Klausel C durch die Typsubstitution ξ bzw. die Termsubstitution σ festgelegt werden und damit weder das instantiierte Pattern-Literal $\sigma_\xi(lit)$ noch die Subgoals $d \in (\sigma_\xi(C) - \sigma_\xi(lit))$ freie Typ- bzw. Termvariablen enthalten.

Beispiel 11.3. Betrachten wir den folgenden Term:

$$\texttt{ordered}(\texttt{merge}(\texttt{msort}(\texttt{dis\_ev}(\texttt{k})), \texttt{msort}(\texttt{dis\_odd}(\texttt{k})))) \,. \tag{11.3}$$

Für die Multimenge

$$\mathcal{Q} = \{\!\{1 * ih_1, 1 * ih_2, 1 * lem\}\!\} \tag{11.4}$$

mit

$$\begin{aligned} \mathcal{C}_{ih_1} &= \{\{\texttt{ordered}(\texttt{msort}(\texttt{dis\_ev}(\texttt{k})))\}\}, \\ \mathcal{C}_{ih_2} &= \{\{\texttt{ordered}(\texttt{msort}(\texttt{dis\_odd}(\texttt{k})))\}\}, \\ \mathcal{C}_{lem} &= \{\{\neg\texttt{ordered}(\texttt{k}'), \neg\texttt{ordered}(\texttt{l}'), \texttt{ordered}(\texttt{merge}(\texttt{k}', \texttt{l}'))\}\} \end{aligned}$$

ergibt sich für den Term (11.3) die folgende symbolische Auswertung:

$$\begin{aligned} \mathcal{U} \vdash\ & \underline{\texttt{ordered}(\texttt{merge}(\texttt{msort}(\texttt{dis\_ev}(\texttt{k})), \texttt{msort}(\texttt{dis\_odd}(\texttt{k}))))} && \rightarrow \\ & \texttt{if}(\underline{\texttt{ordered}(\texttt{msort}(\texttt{div\_ev}(\texttt{k})))}, \texttt{ordered}(\texttt{msort}(\texttt{dis\_odd}(\texttt{k}))), \texttt{false}) && \rightarrow \\ & \underline{\texttt{if}(\texttt{true}, \texttt{ordered}(\texttt{msort}(\texttt{dis\_odd}(\texttt{k}))), \texttt{false})} && \rightarrow \\ & \underline{\texttt{ordered}(\texttt{msort}(\texttt{dis\_odd}(\texttt{k})))} && \rightarrow \\ & \texttt{true}. \end{aligned}$$

Im ersten Auswertungsschritt haben wir mit Hilfe der Regel (11.1) die Klausel der Klauselmenge $\mathcal{C}_{lem}$ angewendet. In den beiden folgenden Anwendungsschritten konnten wir dann die Subgoals

$$\texttt{ordered}(\texttt{msort}(\texttt{div\_ev}(\texttt{k})))$$

und

$$\texttt{ordered}(\texttt{msort}(\texttt{div\_odd}(\texttt{k})))$$

durch die Klauseln der Klauselmengen $\mathcal{C}_{ih_1}$ und $\mathcal{C}_{ih_2}$ zu `true` vereinfachen. Insgesamt haben wir so den Term zu `true` auswertet. □

11.2 Auswertung der Subgoals

Wir klären in diesem Abschnitt die Frage, wie die Subgoals $\sigma_\xi(C) - \sigma_\xi(lit)$ in den Regeln (11.1) und (11.2) symbolisch ausgewertet werden. Die Herausforderung ist hierbei eine effiziente Auswertung der Subgoals durch entsprechende Einschränkung des Suchraums zu definieren ohne die Mächtigkeit der Regeln (11.1) und (11.2) für die in der Praxis auftretenden Fälle spürbar zu beschränken. Wir gehen hierzu wie folgt vor:

- In Abschnitt 11.2.1 definieren wir das Search-Limit für die Subgoals, das die maximale Größe des betrachteten Suchraums festlegt.
- In Abschnitt 11.2.2 gehen wir auf die lokale Hypothesenmenge ein, die zur Auswertung der Subgoals verwendet wird.
- In Abschnitt 11.2.3 klären wir dann die Frage, mit welchen Regeln Prozeduraufrufe in den Subgoals ausgewertet werden dürfen.

- In Abschnitt 11.2.4 definieren wir schließlich den so genannten „Repetition-Filter“. Diese Filter beschränkt die Menge der Quantifikationen, die während der Auswertung der Subgoals verwendet werden dürfen.

11.2.1 Das Search-Limit

Anwendungsbedingung (2) führt in die vorläufigen Regeldefinitionen (11.1) und (11.2) einen unendlichen Suchraum ein, der zu einem unendlichen rekursiven Abstieg während der Überprüfung der Anwendungsbedingung führen kann (vergleiche hierzu auch das Unfold-Limit in Abschnitt 10.1.2). Um den Suchraum zu begrenzen und damit den unendlichen rekursiven Abstieg zu verhindern, enthalten die A-Annotationen das Search-Limit.

Die Search-Limits des Goal-Terms der initialen Sequenz eines Beweisbaums werden vom ✓eriFun-System auf den Wert $s_{\max} = 4^{depth_{\max}}$ gesetzt, wobei $depth_{\max}$ eine von ✓eriFun vorgegebene Konstante bezeichnet. Vor einer Überprüfung der Anwendungsbedingungen der Regeln (11.1) und (11.2) überprüfen wir dann, ob das Search-Limit s des Proof-Goals a größer als 0 ist. Ist dies nicht der Fall, so werden für das Proof-Goal die Regeln (11.1) und (11.2) nicht angewendet. Ist das Search-Limit s größer als 0, so berechnen wir für die Klausel $C \in \bigcup_{q \in \mathcal{Q}} \mathcal{C}_q$ zunächst das neue Search-Limit

$$s_{\text{new}}(|match_{\simeq_{\mathcal{U}}}(lit, a)|, |C|) \tag{11.5}$$

bzw.

$$s_{\text{new}}(|match_{\simeq_{\mathcal{U}}}(\overline{lit}, a)|, |C|)\,. \tag{11.6}$$

Hierbei ist die Funktion s_{new} wie folgt definiert:

$$s_{\text{new}}(m, n) := \left\lfloor \frac{s}{max(n, 4) * m} \right\rfloor . \tag{11.7}$$

Wir überprüfen dann die Anwendungsbedingung (2) der Regeln nur, wenn

$$s_{\text{new}}(|match_{\simeq_{\mathcal{U}}}(lit, a)|, |C|) \geq 1 \quad \text{bzw.} \quad s_{\text{new}}(|match_{\simeq_{\mathcal{U}}}(\overline{lit}, a)|, |C|) \geq 1$$

gilt und annotieren zur Überprüfung der Anwendungsbedingung die Subgoals d mit dem Search-Limit (11.5) bzw. (11.6). Durch diese Definition des Search-Limits erhalten wir für jede Klausel C und jedes Literal $lit \in C$ unabhängig von der Kardinalität der Klausel C und der Anzahl der Matcher von lit bzw. $\overline{lit}$ für a einen annähernd gleich-großen Suchraum. Insbesondere verringert sich für Klauseln mit vielen Literalen die Tiefe des durch die Anwendungsbedingung (2) definierten Suchbaums. Die Verwendung von $max(|C|, 4)$ statt $|C|$ im Quotienten von (11.7) ist notwendig, um für kleine Klauseln unnötig tiefe Suchbäume zu vermeiden. Durch die initiale Annotierung durch $4^{depth_{\max}}$ und die Verwendung von $max(|C|, 4)$ in (11.7) ist sichergestellt, dass der Suchbaum höchstens eine Tiefe von $depth_{\max}$ besitzt.[3]

Insgesamt ergibt sich dann die folgende Modifikation der vorläufigen Auswertungsregeln:

Modifikation (symbolische Auswertung der Subgoals):
Die Regeln (11.1) und (11.2) werden für ein Atom a nur dann angewendet, falls das Search-Limit s des Atoms größer als 0 ist und das neue Search-Limit (11.5) bzw. (11.6) ebenfalls größer als 0 ist. Weiter werden die Subgoals zur symbolischen Auswertung in Anwendungsbedingung (2) mit dem neuen Search-Limit (11.5) bzw. (11.6) annotiert.

[3] In unseren Fallstudien hat sich ein Wert von 4 für $depth_{\max}$ als guter Kompromiss zwischen Effizienz und Mächtigkeit erwiesen.

```
D_dis_ev  =  function dis_ev(k : list[nat]) : list[nat] ⇐
               if(?empty(k),
                  empty
                  if(?empty(tl(k)),
                     empty,
                     add(hd(tl(k)), dis_ev(tl(tl(k))))))

D_dis_odd =  function dis_odd(k : list[nat]) : list[nat] ⇐
               if(?empty(k),
                  empty
                  if(?empty(tl(k)),
                     k,
                     add(hd(k), dis_ev(tl(tl(k))))))

D_merge   =  function merge(k : list[nat], l : list[nat]) : list[nat] ⇐
               if(?empty(k),
                  l
                  if(?empty(l),
                     k,
                     if(hd(k)>hd(l),
                        add(hd(l), merge(k, tl(l)))
                        add(hd(k), merge(tl(k), l)))))
```

Abbildung 11.1: Prozedurdefinitionen für `dis_ev`, `dis_odd` und `merge`.

11.2.2 Verwendung des negierten Proof-Goals

Angenommen wir möchten auf ein Atom a eine Klausel $C = \{a, b, c\}$ einer Quantifikation $q \in \mathcal{Q}$ anwenden. Wir erhalten dann für die Regel (11.1) die Subgoals b und c, welche wir mit der A-Umgebung $\mathcal{U}$ zu `false` auswerten müssen. Während dieser Auswertung ist es zulässig $a =$ `false` anzunehmen. Können wir nämlich die Subgoals b und c unter dieser Annahme zu `false` auswerten, so haben wir insgesamt die Ungültigkeit der Klausel $\{a, b, c\}$ nachgewiesen. Da wir jedoch während der symbolischen Auswertung ausschließlich bereits verifizierte Quantifikationen bzw. Quantifikationen von Induktionshypothesen verwenden, haben wir einen Widerspruch generiert. Das bedeutet, die Annahme $a =$ `false` muss falsch sein und das Atom a dementsprechend gültig.

Mit gleichen Argumentationen ist es für die Auswertung der Subgoals in der Regel (11.2) zulässig die Gültigkeit des Atoms a anzunehmen. Insgesamt führen diese Überlegungen zu der folgenden Modifikation:

Modifikation (Erweiterung der lokalen Hypothesen):
Zur Auswertung der Subgoals in der Regel (11.1) erweitern wir die lokale Hypothesenmenge H_{L} um das negierte Proof-Goal $\neg a$. Zur Auswertung der Subgoals in der Regel (11.2) erweitern wir die lokale Hypothesenmenge H_{L} um das Proof-Goal a.

Wir illustrieren die Nützlichkeit der Modifikation anhand eines Beispiels.

Beispiel 11.4. Sei das Programm P geben durch $\langle D_{\texttt{list}}, D_{\texttt{member}}, D_{\texttt{dis\_odd}}, D_{\texttt{dis\_ev}}, D_{\texttt{merge}}\rangle$, wobei $D_{\texttt{list}}$ die Typoperatordefinition aus Abbildung 3.1 bezeichnet und $D_{\texttt{member}}, D_{\texttt{dis\_odd}}, D_{\texttt{dis\_ev}}, D_{\texttt{merge}}$ die Prozedurdefinitionen aus den Abbil-

$$
\begin{array}{lll}
\mathcal{U} & \vdash & \texttt{if}(M, \underline{\texttt{member}(\texttt{n},\texttt{l})}, \texttt{true}) \qquad (1)\uparrow \\
 & & \texttt{if}(M, \texttt{if}(\texttt{member}(\texttt{n},\texttt{l})\texttt{true}, \underline{\texttt{member}(\texttt{n},\texttt{dis\_odd}(\texttt{l}))}, \texttt{true}) \qquad (2)\uparrow \\
 & & \texttt{if}(M, \texttt{if}(\texttt{member}(\texttt{n},\texttt{l})\texttt{true}, \texttt{if}(\texttt{member}(\texttt{n},\texttt{dis\_odd}(\texttt{l})), \texttt{true}, \texttt{if}(\underline{\texttt{member}(\texttt{n},\texttt{dis\_ev}(\texttt{l}))}, \texttt{false}, M))), \texttt{true}) \qquad (3)\uparrow \\
 & & \texttt{if}(M, \texttt{if}(\texttt{member}(\texttt{n},\texttt{l})\texttt{true}, \texttt{if}(\texttt{member}(\texttt{n},\texttt{dis\_odd}(\texttt{l})), \texttt{true}, \texttt{if}(\texttt{member}(\texttt{n},\texttt{dis\_ev}(\texttt{l})), \underline{\texttt{member}(\texttt{n},\texttt{l})}, \texttt{false}), \texttt{false}, M))), \texttt{true}) \qquad (4)\uparrow \\
 & & \texttt{if}(M, \texttt{if}(\texttt{member}(\texttt{n},\texttt{l})\texttt{true}, \texttt{if}(\texttt{member}(\texttt{n},\texttt{dis\_odd}(\texttt{l})), \texttt{true}, \texttt{if}(\underline{\texttt{if}(\texttt{member}(\texttt{n},\texttt{dis\_ev}(\texttt{l})), \texttt{false}, \texttt{false})}, \texttt{false}, M))), \texttt{true}) \qquad (6)\uparrow \\
 & & \texttt{if}(M, \texttt{if}(\texttt{member}(\texttt{n},\texttt{l})\texttt{true}, \texttt{if}(\texttt{member}(\texttt{n},\texttt{dis\_odd}(\texttt{l})), \texttt{true}, \underline{\texttt{if}(\texttt{false}, \texttt{false}, M)})), \texttt{true}) \qquad (6)\uparrow \\
 & & \texttt{if}(M, \texttt{if}(\texttt{member}(\texttt{n},\texttt{l})\texttt{true}, \texttt{if}(\texttt{member}(\texttt{n},\texttt{dis\_odd}(\texttt{l})), \texttt{true}, \underline{M})), \texttt{true}) \qquad (7)\uparrow \\
 & & \texttt{if}(M, \texttt{if}(\texttt{member}(\texttt{n},\texttt{l})\texttt{true}, \underline{\texttt{if}(\texttt{member}(\texttt{n},\texttt{dis\_odd}(\texttt{l})), \texttt{true}, \texttt{true})}), \texttt{true}) \qquad (8)\uparrow \\
 & & \texttt{if}(M, \underline{\texttt{if}(\texttt{member}(\texttt{n},\texttt{l})\texttt{true}, \texttt{true})}, \texttt{true}) \qquad (9)\uparrow \\
 & & \underline{\texttt{if}(M, \texttt{true}, \texttt{true})} \qquad (10)\uparrow \\
 & & \texttt{true}.
\end{array}
$$

Abbildung 11.2: Symbolische Auswertung des Terms (11.8). Hierbei bezeichnet M den Teilterm $\texttt{member}(\texttt{n}, \texttt{merge}(\texttt{dis\_ev}(\texttt{l}), \texttt{dis\_odd}(\texttt{l})))$.

dungen 10.1 und 11.1. Wir wollen nun den Term

$$\texttt{if}(\texttt{member}(\texttt{n}, \texttt{merge}(\texttt{div\_ev}(\texttt{l}), \texttt{div\_odd}(\texttt{l}))), \texttt{member}(\texttt{n}, \texttt{l}), \texttt{true}) \tag{11.8}$$

symbolisch auswerten. Zur symbolischen Auswertung des Terms verwenden wir die folgenden Lemmata, d.h. wir gehen davon aus, dass die Gültigkeit dieser Lemmata bereits bekannt ist:

$$\begin{array}{l}\texttt{lemma member\_dis\_odd\_entails\_member} \Leftarrow \texttt{all n} : \texttt{nat}, \texttt{l} : \texttt{list}[\texttt{nat}] \\ \quad \texttt{if}(\texttt{member}(\texttt{n}, \texttt{dis\_odd}(\texttt{l})), \texttt{member}(\texttt{n}, \texttt{l}), \texttt{true})\end{array}$$

$$\begin{array}{l}\texttt{lemma member\_dis\_ev\_entails\_member} \Leftarrow \texttt{all n} : \texttt{nat}, \texttt{l} : \texttt{list}[\texttt{nat}] \\ \quad \texttt{if}(\texttt{member}(\texttt{n}, \texttt{dis\_ev}(\texttt{l})), \texttt{member}(\texttt{n}, \texttt{l}), \texttt{true})\end{array}$$

$$\begin{array}{l}\texttt{lemma member\_merge} \Leftarrow \texttt{all n} : \texttt{nat}, \texttt{l}, \texttt{k} : \texttt{list}[\texttt{nat}] \\ \quad \texttt{if}(\texttt{member}(\texttt{n}, \texttt{merge}(\texttt{l}, \texttt{k})), \texttt{if}(\texttt{member}(\texttt{n}, \texttt{l}), \texttt{true}, \texttt{member}(\texttt{n}, \texttt{k})), \texttt{true})\end{array}$$

Für die Lemmata ergeben sich die folgenden Klauseln:

$$C_1 = \{\neg\texttt{member}(\texttt{n}, \texttt{dis\_odd}(\texttt{l}')), \texttt{member}(\texttt{n}', \texttt{l}')\}$$

$$C_2 = \{\neg\texttt{member}(\texttt{n}, \texttt{dis\_ev}(\texttt{l})'), \texttt{member}(\texttt{n}', \texttt{l}')\}$$

$$C_3 = \{\neg\texttt{member}(\texttt{n}', \texttt{merge}(\texttt{l}', \texttt{k}')), \texttt{member}(\texttt{n}', \texttt{l}'), \texttt{member}(\texttt{n}', \texttt{k}')\}$$

Die symbolische Auswertung des Terms (11.8) ist in Abbildung 11.2 dargestellt. Es ist zu beachten, dass wir zur Visualisierung des negierten Proof-Goals bei Anwendung der Regel (11.1) bzw. (11.2) das Proof-Goal a nicht durch

$$NOR(\sigma_\xi(C) - \sigma_\xi(lit)) \quad \text{bzw.} \quad OR(\sigma_\xi(C) - \sigma_\xi(lit))$$

ersetzen, sondern durch den Term

$$\texttt{if}(a, \texttt{true}, NOR(\sigma_\xi(C) - \sigma_\xi(lit))) \quad \text{bzw.} \quad \texttt{if}(a, OR(\sigma_\xi(C) - \sigma_\xi(lit)), \texttt{false}).$$

Weiter ist zu beachten, dass die Anwendung der Klausel C_2 in Auswertungsschritt (3) nur aufgrund des negierten Proof-Goals $\texttt{member}(\texttt{n}, \texttt{l})$ aus Auswertungsschritt (1) möglich ist. □

Wir wenden Regel (11.1) vor Regel (11.2) an. Dies führt dazu, dass manche offensichtliche symbolische Auswertungen blockiert sind. Das Problem liegt hierbei nicht an der konkreten Reihenfolge der Regeln - ein Vertauschen der Reihenfolge der Regeln (11.1) und (11.2) führt zu analogen Problemen - sondern vielmehr daran, dass wir die Regeln in einer festen Reihenfolge anwenden. Betrachten wir dazu ein Beispiel.

Beispiel 11.5. Nehmen wir an, dass wir den Term a mit Hilfe der Klauseln

$$\begin{aligned} C_1 &= \{a, b\}, \\ C_2 &= \{a, \neg b\} \end{aligned}$$

zu `true` auswerten möchten. Eine Anwendung der Klausel C_1 auf den Term a mit Hilfe der Regel (11.1), unter Berücksichtigung unserer Anforderungen, scheint hierbei möglich, da das erzeugte Subgoal b unter der Annahme $\overline{a}$ mit Hilfe der Regel (11.2) und der Klausel C_2 zu `false` ausgewertet werden kann:[4]

[4] $\mathcal{U}'$ bezeichnet hierbei die A-Umgebung mit $H_\mathrm{L} = \{\neg a\}$.

$$\begin{array}{lll} \mathcal{U}' \vdash \underline{b} & \rightarrow \\ \underline{a} & \rightarrow \\ \texttt{false}. \end{array}$$

Auf das Subgoal b ist jedoch auch die Regel (11.1) mit der Klausel C_1 anwendbar:

$$\begin{array}{lll} \mathcal{U}' \vdash \underline{b} & \rightarrow \\ \neg\underline{a} & \rightarrow \\ \underline{\neg\texttt{false}} & \rightarrow \\ \texttt{true}. \end{array}$$

Da wir die Regel (11.1) vor der Regel (11.2) anwenden, werten wir somit das Subgoal b zu `true` und nicht wie erwünscht zu `false` aus. Wir können daher die Klausel C_1 nicht auf den Term a anwenden und damit auch nicht zu `true` vereinfachen. Das bedeutet, dass die symbolische Auswertung von a blockiert ist. □

Die Ursache des in Beispiel 11.5 geschilderten Problems ist, dass wir auf Subgoal b, welches zu `false` ausgewertet werden soll, Regel (11.1) anwenden, obwohl wir wissen, dass bei einer erfolgreichen Anwendung dieser Regel der Term zu `true` auswertet wird und somit die Anwendung der Klausel C_1 verhindert wird. Eine Anwendung der Regel (11.1) ist daher für die Auswertung des Subgoals b kontraproduktiv. Wir können solche kontraproduktiven Anwendungen der Regeln (11.1) und (11.2) während der symbolischen Auswertung von Subgoals verhindern, indem wir die Subgoals mit Polaritäten annotieren. Die Polarität legt fest, ob das Subgoal zu `true` oder `false` ausgewertet werden soll und wir wenden die Regeln (11.1) und (11.2) auf ein Subgoal nur noch dann an, wenn die Anwendung der Regeln der Polarität des Subgoals nicht widerspricht. Zur Annotierung mit Polaritäten definieren wir die folgende Funktion. Hierbei bezeichnet a immer ein Atom:[5]

$$\begin{array}{rcl} \mathbf{pol}_p(a^{\,p'}) & := & \mathbf{pol}(a^{\,p}) \\ \mathbf{pol}_\odot(\texttt{if}^{\,A}(a^{\,p}, \texttt{false}, \texttt{true})) & := & \texttt{if}^{\,A}(a^{\,\odot}, \texttt{false}, \texttt{true}), \\ \mathbf{pol}_\ominus(\texttt{if}^{\,A}(a^{\,p}, \texttt{false}, \texttt{true})) & := & \texttt{if}^{\,A}(a^{\,\oplus}, \texttt{false}, \texttt{true}), \\ \mathbf{pol}_\oplus(\texttt{if}^{\,A}(a^{\,p}, \texttt{false}, \texttt{true})) & := & \texttt{if}^{\,A}(a^{\,\ominus}, \texttt{false}, \texttt{true}). \end{array}$$

Regel (11.1) wenden wir jetzt nur dann auf ein Proof-Goal a an, wenn a mit einer nicht-negativen Polarität annotiert ist. Regel (11.2) hingegen wenden wir nur an, wenn a mit einer nicht-positiven Polarität annotiert ist. Insgesamt ergibt sich somit die folgende Modifikation:

Modifikation (Polaritäten):

Zur symbolischen Auswertung der Subgoals in der Anwendungsbedingung (2) wird die Polarität der Subgoals mit Hilfe der Funktion $\mathbf{pol}_\ominus$ gesetzt. Weiter werden die Regeln (11.1) und (11.2) nur noch dann angewendet, wenn das auszuwertende Atom a eine nicht-negative bzw. nicht-positive Polarität hat.

In Beispiel 11.5 annotieren wir dementsprechend das Subgoal b mit $\ominus$ und wenden daher Regel (11.1) nicht auf dieses Subgoal an. Folglich wird Regel (11.2) angewendet und somit das Subgoal zu `false` vereinfacht. Damit können wir auf den Term a die Klausel C_1 anwenden und so a zu `true` vereinfachen.

[5] Der Übersichtlichkeit halber stellen wir in der Definition von **pol** ausschließlich die Polaritäten der A-Annotationen dar.

11.2.3 Auswertung von Prozeduraufrufen

Um eine effiziente Auswertung der Subgoals in der Anwendungsbedingung (2) zu gewährleisten, müssen wir die Anwendung der „*Unfold*“-Regeln verbieten.

Modifikation (Prozeduraufrufe in Subgoals):
Das Unfold-Limit der Subgoals wird zur symbolischen Auswertung in Anwendungsbedingung (2) auf 0 gesetzt.

Weiter hat sich durch unsere Fallstudien gezeigt, dass eine Auswertung von Prozeduraufrufen in den Subgoals durch die Regeln

„*Execute procedure call (recursive cases)*“,
„*Execute procedure call (non recursive cases)*“,
„*Execute procedure call (additional case analysis)*“

und ihren kommutierten Versionen für eine effiziente Auswertung der Subgoals ebenfalls zu aufwendig ist. Wir müssen daher während der Vereinfachung der Subgoals auch auf diese Regeln verzichten. Dies hat insbesondere zur Konsequenz, dass Prozeduraufrufe rekursiv-definierter Prozeduren ausschließlich mit der Regel „*Execute procedure call (constructor ground terms)*“ ausgewertet werden. Dies reicht jedoch für eine erfolgreiche Auswertung der Subgoals häufig nicht aus. Betrachten wir dazu ein Beispiel:

Beispiel 11.6. Sei P das Programm $\langle D_{\texttt{IF.Expr}}, D_{\texttt{IF.depth}}\rangle$, wobei $D_{\texttt{IF.Expr}}$ und $D_{\texttt{IF.depth}}$ die folgenden Typoperator- und Prozedurdefinitionen bezeichnen:

$$
\begin{array}{rcl}
D_{\texttt{IF.Expr}} & = & \texttt{structure IF.Expr} \Leftarrow \\
& & \quad \texttt{F}, \\
& & \quad \texttt{T}, \\
& & \quad \texttt{prop(number : nat)}, \\
& & \quad \texttt{IF(test : IF.Expr, left : IF.Expr, right : IF.Expr)}, \\
 & & \\
D_{\texttt{IF.depth}} & = & \texttt{function IF.depth(x : IF.Expr) : nat} \Leftarrow \\
& & \quad \texttt{if(?IF(x),} \\
& & \quad\quad \texttt{succ(IF.depth(test(x))),} \\
& & \quad\quad \texttt{0)}.
\end{array}
$$

Sei weiter die Multimenge $\mathcal{Q} = \{\!\{1 * q_{\texttt{>\_total}}, 1 * q_{\texttt{>\_asymmetric}}, 1 * q_{\texttt{>\_succ*}}\}\!\}$ mit

$$
\begin{array}{rcl}
\mathcal{C}_{q_{\texttt{>\_total}}} & = & \{\{\texttt{x>y}, \texttt{y>x}, \texttt{x=y}\}\}, \\
\mathcal{C}_{q_{\texttt{>\_asymmetric}}} & = & \{\{\neg\texttt{x>y}, \neg\texttt{y>x}\}\}, \\
\mathcal{C}_{q_{\texttt{>\_succ*}}} & = & \{\{\neg\texttt{x=y}, \texttt{succ(x)>y}\}\}
\end{array}
$$

gegeben. Betrachten wir nun den Term

```
if(?IF(x),
   if(?IF(test(x)),
      pred(IF.depth(x))>IF.depth(test(test(x))),          (11.9)
      true),
   true).
```

Diesen Term können wir mit Hilfe der Multimenge $\mathcal{Q}$ nur dann auswerten, falls wir während der Auswertung von Subgoals Prozeduraufrufe von `IF.depth` auswerten. Die sich dadurch ergebende symbolische Auswertung ist in Abbildung 11.3

dargestellt. Hierbei haben wir Teilterme, die Subgoals repräsentieren kursiv dargestellt. Im Auswertungsschritt (1) der symbolischen Auswertung wenden wir die Regel (11.2) mit der Klauselmenge $\mathcal{C}_{q>\_\texttt{total}}$ an und erzeugen dadurch die Subgoals

$$\texttt{IF.depth(test(test(x)))>pred(IF.depth(x))}$$

und

$$\texttt{pred(IF.depth(x))=IF.depth(test(test(x)))}.$$

Für das erste Subgoal werten wir dann im zweiten Auswertungsschritt den Prozeduraufruf von `IF.depth` aus. Durch Anwendung der Klauseln von $\mathcal{C}_{q>\_\texttt{asymmetric}}$ und $\mathcal{C}_{q>\_\texttt{succ*}}$ durch die Regeln (11.2) und (11.1) in den Auswertungsschritten (3) und (4) vereinfachen wir anschließend das erste Subgoal zu `true`. Ohne Auswertung des Prozeduraufrufs in Auswertungsschritt (2) wäre eine Anwendung der Klauselmengen $\mathcal{C}_{q>\_\texttt{asymmetric}}$ nicht möglich. Die Auswertung des Prozeduraufrufs in Auswertungsschritt (9) ist ebenfalls erforderlich, da wir ansonsten die Regel „*Constructor uniqueness*“ nicht anwenden können. □

Um Auswertungen von Prozeduraufrufen rekursiv-definierter Prozeduren wie in Beispiel 11.6 zu ermöglichen, erlauben wir für die Auswertung der Subgoals die Verwendung der Regel „*Execute procedure call (no additional case analysis)*“ auch für rekursiv-definierte Prozeduren. Das bedeutet, wir werten Prozeduraufrufe nicht nur mit den Regeln „*Execute procedure call (constructor ground terms)*“ und „*Execute procedure call (no additional case analysis)*“ sowie mit ihren kommutierten Versionen aus, sondern auch mit Hilfe der folgenden Regeln:

49. Execute procedure call (recursive – no additional case analysis)

$$\frac{f^{\,A}(t_1,\ldots,t_n)}{\sigma_{\xi_{f(\ldots)}}(\mathsf{s}^*_f(\mathsf{un}^{u,s,0}_f(R^*_f)))},$$

	Label	S-Lim.	U-Lim.
A:	l	s	u

falls $f \in \Sigma^{\mathrm{rec}}(P) \cap \Sigma$, $\sigma = \{x_1/\mathsf{a}_{A_0}(t_1), \ldots, x_n/\mathsf{a}_{A_0}(t_n)\}$ und die folgende Bedingung erfüllt ist:

> Es gilt $l \notin C$ und es existiert ein $\pi \in \mathcal{P}os_{\mathrm{E}}(f)$, so dass für alle $b \in cond(\pi, \sigma(|R_f|_\pi))$ gilt $\mathcal{U} \vdash \mathsf{a}_{A_{\mathrm{top}}}(b) \to \texttt{true}$.

50. Execute procedure call (recursive – no additional case analysis)*

$$\frac{f^{\,A}(t_1,t_2)}{\sigma_{\xi_{f(\ldots)}}(\mathsf{s}^*_f(\mathsf{un}^{u,s,0}_f(R^*_f)))}$$

	Label	S-Lim.	U-Lim.
A:	l	s	u

falls $f \in \Sigma \cap \Sigma^{\mathrm{rec}}(P) \cap \Sigma_{\mathrm{C}}$, $\sigma = \{x_1/\mathsf{a}_{A_0}(t_2), x_2/\mathsf{a}_{A_0}(t_1)\}$ und die folgende Bedingung erfüllt ist:

> Es gilt $l \notin C$ und es existiert ein $\pi \in \mathcal{P}os_{\mathrm{E}}(f)$, so dass für alle $b \in cond(\pi, \sigma(|R_f|_\pi))$ gilt $\mathcal{U} \vdash \mathsf{a}_{A_{\mathrm{top}}}(b) \to \texttt{true}$.

Aufbauend auf diesen Regeln können wir die folgende Modifikation definieren:

$$
\begin{array}{lll}
\mathcal{U} \vdash & \texttt{if(?IF(x), if(?IF(test(x)), }\underline{\texttt{pred(IF.depth(x))>IF.depth(test(test(x)))}}\texttt{, true), true)} & \overset{(1)}{\rightarrow} \\
& \texttt{if(}\ldots\texttt{, if(}\neg(\underline{\mathit{IF.depth}(\mathit{test}(\mathit{test}(x)))>\mathit{pred}(\mathit{IF.depth}(x))}), \neg(\mathit{pred}(\mathit{IF.depth}(x))=\mathit{IF.depth}(\mathit{test}(\mathit{test}(x)))\texttt{, false), true), true)} & \overset{(2)}{\rightarrow} \\
& \vdots & \vdots \\
& \texttt{if(}\ldots\texttt{, if(}\neg(\underline{\mathit{IF.depth}(\mathit{test}(\mathit{test}(x)))>\mathit{succ}(\mathit{IF.depth}(\mathit{test}(\mathit{test}(x))))}), \neg(\mathit{pred}(\mathit{IF.depth}(x))=\mathit{IF.depth}(\mathit{test}(\mathit{test}(x)))\texttt{, false), true), true)} & \overset{(3)}{\rightarrow} \\
& \texttt{if(}\ldots\texttt{, if(}\neg(\neg(\underline{\mathit{succ}(\mathit{IF.depth}(\mathit{test}(\mathit{test}(x))))>\mathit{IF.depth}(\mathit{test}(\mathit{test}(x))))})), \neg(\mathit{pred}(\mathit{IF.depth}(x))=\mathit{IF.depth}(\mathit{test}(\mathit{test}(x)))\texttt{, false), true), true)} & \overset{(4)}{\rightarrow} \\
& \texttt{if(}\ldots\texttt{, if(}\neg(\neg(\underline{\mathit{IF.depth}(\mathit{test}(\mathit{test}(x)))=\mathit{IF.depth}(\mathit{test}(\mathit{test}(x)))}), \neg(\mathit{pred}(\mathit{IF.depth}(x))=\mathit{IF.depth}(\mathit{test}(\mathit{test}(x)))\texttt{, false), true), true)} & \overset{(5)}{\rightarrow} \\
& \texttt{if(}\ldots\texttt{, if(}\neg(\underline{\neg\mathit{true}})), \neg(\mathit{pred}(\mathit{IF.depth}(x))=\mathit{IF.depth}(\mathit{test}(\mathit{test}(x)))\texttt{, false), true), true)} & \overset{(6)}{\rightarrow} \\
& \texttt{if(}\ldots\texttt{, if(}\underline{\neg\mathit{false}}\texttt{, }\neg(\mathit{pred}(\mathit{IF.depth}(x))=\mathit{IF.depth}(\mathit{test}(\mathit{test}(x)))\texttt{, false), true), true)} & \overset{(7)}{\rightarrow} \\
& \texttt{if(}\ldots\texttt{, }\underline{\mathit{if}(\mathit{true}}\texttt{, }\neg(\mathit{pred}(\mathit{IF.depth}(x))=\mathit{IF.depth}(\mathit{test}(\mathit{test}(x)))\texttt{, }\underline{\mathit{false}}\texttt{), true), true)} & \overset{(8)}{\rightarrow} \\
& \texttt{if(?IF(x), if(?IF(test(x)), }\neg(\mathit{pred}(\underline{\mathit{IF.depth}(x)})=\mathit{IF.depth}(\mathit{test}(\mathit{test}(x)))\texttt{, true), true)} & \overset{(9)}{\rightarrow} \\
& \vdots & \vdots \\
& \texttt{if(?IF(x), if(?IF(test(x)), }\neg(\underline{\mathit{succ}(\mathit{IF.depth}(\mathit{test}(\mathit{test}(x))))=\mathit{IF.depth}(\mathit{test}(\mathit{test}(x))))})\texttt{, true), true)} & \overset{(10)}{\rightarrow} \\
& \texttt{if(?IF(x), if(?IF(test(x)), }\underline{\neg\mathit{false}}\texttt{, true), true)} & \overset{(11)}{\rightarrow} \\
& \texttt{if(?IF(x), }\underline{\texttt{if(?IF(test(x)), }\mathit{true}\texttt{, true)}}\texttt{, true)} & \overset{(12)}{\rightarrow} \\
& \underline{\texttt{if(?IF(x), true, true)}} & \overset{(13)}{\rightarrow} \\
& \texttt{true} & \rightarrow
\end{array}
$$

Abbildung 11.3: Symbolische Auswertung des Terms (11.9) aus Beispiel 11.6 unter Berücksichtigung der Multimenge $\mathcal{Q}$. In den Auswertungsschritten (2) und (9) werden die Prozeduraufrufe in den Subgoals ausgewertet.

Modifikation (Prozeduraufrufe in Subgoals):
Prozeduraufrufe in Subgoals werden ausschließlich durch die folgenden „*Execute*"-Regeln und ihrer kommutierten Versionen ausgewertet:

„*Execute procedure call (constructor ground terms)*",
„*Execute procedure call (no additional case analysis)*",
„*Execute procedure call (recursive – no additional case analysis)*".

Um die Regelmenge der aktuellen A-Umgebung entsprechend anzupassen, wird die folgende Funktion verwendet:

$$\mathit{prim\text{-}execute}(\mathcal{R}) := (\mathcal{R}_0 \setminus \{41, \ldots, 52\}) \cup \{49, 50\}\,.$$

Anmerkung 11.7. Da sich die Regel „*Execute procedure call (recursive – no additional case analysis)*" zur Auswertung von Subgoals als nützlich erweist, stellt sich die Frage, warum wir diese Regel nicht generell während der symbolischen Auswertung verwenden. Der Grund hierfür ist, dass diese Regel im Zusammenspiel mit der Regel „*Execute procedure call (recursive cases)*" zu Endlos-Auswertungen führen kann (siehe Beispiel F.8 in Anhang F). □

11.2.4 Repetition-Filter

Die Subgoals werden in der Anwendungsbedingung (2) der Regeln (11.1) und (11.2) durch die aktuelle A-Umgebung $\mathcal{U}$ ausgewertet. Das bedeutet insbesondere, dass die Subgoals unter der gleichen Multimenge $\mathcal{Q}$ von Quantifikationen ausgewertet werden, wie das Proof-Goal a. Der dadurch festgelegte Suchraum ist trotz der in den Abschnitten 11.2.1 und 11.2.3 definierten Modifikationen zu groß, um eine effiziente Anwendung von Quantifikationen bzw. Klauseln zu gewährleisten. Wir definieren daher die folgende zusätzliche Modifikation:

Modifikation (Repetition-Filter):
Aus der Multimenge $\mathcal{Q}$ der Quantifikationen wird für die symbolische Auswertung der Subgoals die Quantifikation q entfernt, deren assoziierte Klauselmenge $\mathcal{C}_q$ die anzuwendende Klausel C enthält. Ob die Klausel C zur Auswertung der Subgoals verwendet werden darf, hängt dann davon ab, wie häufig eine Quantifikation q mit $C \in \mathcal{C}_q$ in der Multimenge $\mathcal{Q}$ der A-Umgebung ursprünglich enthalten ist.

Das ✓eriFun-System berechnet zur symbolischen Auswertung eines Terms t die Anzahl der Quantifikationen in $\mathcal{Q}$ wie folgt:

- **Induktionshypothesen**: Die Quantifikationen der Induktionshypothesen sind in der Multimenge grundsätzlich nur einmal enthalten. Das bedeutet, dass zur Auswertung von Subgoals die durch eine Induktionshypothesen erzeugt wurden, die Induktionshypothese selbst nicht verwendet wird. Da dies in keiner unserer Fallstudien notwendig war, bedeutet diese Begrenzung aus praktischer Sicht keine Einschränkung der Mächtigkeit des Auswertungskalküls.

- **Transitivität**: Für Lemmata der Form

$$\begin{array}{l}\texttt{lemma}\ \mathit{lem} \Leftarrow \texttt{all}\ x, y, z : \tau \\ \quad \texttt{if}(f(x, y), \texttt{if}(f(y, z), f(x, z), \texttt{true}), \texttt{true})\end{array}$$

sind die entsprechenden Quantifikationen $depth_{\max}$-mal in der Multimenge $\mathcal{Q}$ enthalten. Da $depth_{\max}$ die maximale Suchtiefe für die Regeln (11.1) und (11.2) festlegt, bedeutet das, dass für die Transitivität keine Beschränkungen bzgl. der Verwendung in Subgoals existieren. Dies ist notwendig, um lange „Transitivitätsketten" erfolgreich auswerten zu können. Beispielsweise können wir so einen Term wie

$$\texttt{if(x>y, if(y>z, if(z>u, x>u, true), true), true)}$$

mit Hilfe des Auswertungskalküls zu `true` auswerten.

- **Andere Quantifikationen**: Für alle anderen Lemmata werden die Quantifikationen zweimal in die Multimenge eingefügt. Dadurch ergibt sich in Kombination mit den anderen Optimierungen eine ausreichende Beschränkung des Suchraums.

11.3 Neue Vorläufige Regeldefinitionen

Wir fügen nun die in Abschnitt 11.2 definierten Modifikationen zusammen und definieren neue vorläufige Auswertungsregeln:

$$\left.\begin{array}{l} \dfrac{a^{\,A}}{\texttt{if}^{\,A_{\text{std}}}(\mathbf{a}_{A_0}(a), \texttt{true}, \mathbf{env}_{\mathcal{U}'}(NOR(d_1,\ldots,d_n)))}, \quad \begin{array}{|c|c|}\hline & \text{S-Lim.} \\ \hline A: & s \\ \hline \end{array} \\ \text{falls } s > 0 \text{ und } \mathbf{e}(a) \in \mathcal{A}t(P) \text{ gilt. Weiter muss ein } C \in \bigcup_{q\in\mathcal{Q}} \mathcal{C}_q \\ \text{existieren, so dass für ein } a' \in \mathit{struct\text{-}set}_P(\mathbf{e}(a)) \text{ und ein } \sigma_\xi \in \\ \mathit{match}_{\simeq_{\mathcal{U}}}(lit, a') \text{ die folgenden Bedingungen erfüllt sind:} \\ (1)\ \ s_{\text{new}} = s_{\text{new}}(|\mathit{match}_{\simeq_{\mathcal{U}}}(lit,a)|, |C|) \text{ und } s_{\text{new}} \geq 1. \\ (2)\ \ \mathit{tv}'(C) \subseteq \mathit{tv}'(lit) \text{ und } \mathit{fv}'(\xi(C)) \subseteq \mathit{fv}'(\xi(lit)). \\ (3)\ \ \{d_1,\ldots,d_n\} = \mathbf{pol}_{\ominus}(\mathbf{a}_{A'}(\sigma_\xi(C) - \sigma_\xi(lit))). \\ (4)\ \ \forall i \in \{1,\ldots,n\} \text{ gilt } \mathcal{U}' \vdash d_i \rightarrow^{!} \texttt{false} \\ \text{Hierbei sind die A-Umgebung } \mathcal{U}' \text{ und die A-Annotation } A' \text{ wie folgt} \\ \text{gegeben:} \end{array}\right\} \quad (11.10)$$

	Hyp_{L}	Quant.	Reg.
$\mathcal{U}'$:	$H_{\text{L}} \cup \{\overline{\mathbf{e}(a)}\}$	$\mathcal{Q} \setminus \{1 * q\}$	$\mathit{prim\text{-}execute}(\mathcal{R})$

A_{std}	S-Lim.
A':	s_{new}

$$\frac{a^{\,A}}{\texttt{if}^{\,A_{\mathrm{std}}}(\mathbf{a}_{A_0}(a), \mathbf{env}_{\mathcal{U}'}(OR(d_1,\ldots,d_n)), \texttt{false})}, \qquad \frac{}{\begin{array}{ll}\multicolumn{2}{r}{\text{S-Lim.}}\\ A\text{:} & s\end{array}}$$

falls $s > 0$ und $\mathbf{e}(a) \in \mathcal{A}t(P)$ gilt. Weiter muss ein $C \in \bigcup_{q\in\mathcal{Q}} \mathcal{C}_q$ existieren, so dass für ein $a' \in \mathit{struct\text{-}set}_P(\mathbf{e}(a))$ und ein $\sigma_\xi \in \mathit{match}_{\simeq_\mathcal{U}}(\mathit{lit}, a')$ die folgenden Bedingungen erfüllt sind:

(1) $s_{\mathrm{new}} = s_{\mathrm{new}}(|\mathit{match}_{\simeq_\mathcal{U}}(\mathit{lit}, a)|, |C|)$ und $s_{\mathrm{new}} \geq 1$.

(2) $\mathit{tv}'(C) \subseteq \mathit{tv}'(\mathit{lit})$ und $\mathit{fv}'(\xi(C)) \subseteq \mathit{fv}'(\xi(\mathit{lit}))$.

(3) $\{d_1,\ldots,d_n\} = \mathbf{pol}_{\ominus}(\mathbf{a}_{A'}(\sigma_\xi(C) - \sigma_\xi(\mathit{lit})))$.

(4) $\forall i \in \{1,\ldots,n\}$ gilt $\mathcal{U}' \vdash d_i \rightarrow^{!} \texttt{false}$

(11.11)

Hierbei sind die A-Umgebung $\mathcal{U}'$ und die A-Annotation A' wie folgt gegeben:

	$\mathrm{Hyp}_{\mathrm{L}}$	Quant.	Reg.
$\mathcal{U}'$:	$H_{\mathrm{L}} \cup \{\mathbf{e}(a)\}$	$\mathcal{Q} \setminus \{1 * q\}$	$\mathit{prim\text{-}execute}(\mathcal{R})$

A_{std}	S-Lim.
A':	s_{new}

Die einzelnen Anforderungen des Abschnitts 11.2 werden durch die Regeln wie folgt realisiert:

- Die Anforderung aus Abschnitt 11.2.1 bzgl. des Search-Limits wird in den Regeln durch die Anwendungsbedingung (1) und die Verwendung der Annotation A' gewährleistet. Durch die Anwendungsbedingung (1) wird das neue Search-Limit berechnet und durch die Annotation A' in den symbolischen Auswertungen der Subgoals verwendet.

- Die Anforderung aus Abschnitt 11.2.2 bzgl. der Verwendung des negierten Proof-Goals als lokale Hypothese wird durch die Festlegung der lokalen Hypothesen der A-Umgebung $\mathcal{U}'$ realisiert. Die Polaritäten der Subgoals werden in der Anwendungsbedingung (3) gesetzt.

- Die Anforderungen aus Abschnitt 11.2.3 bzgl. der Auswertung von Prozeduraufrufen werden wie folgt realisiert: Durch die Definition der Regelmenge der A-Umgebung $\mathcal{U}'$ wird die Verwendung der „*Execute*“-Regeln für die symbolische Auswertung der Subgoals entsprechend beschränkt. Durch die Ableitung der Annotation A' von der Annotation A_{std} wird das Unfold-Limit der Subgoals auf 0 gesetzt.

- Die Anforderung aus Abschnitt 11.2.4 bzgl. der Beschränkung der Quantifikationen wird durch die Elimination der Quantifikation q aus der Multimenge $\mathcal{Q}$ in der Definition der A-Umgebung $\mathcal{U}'$ sichergestellt.

Die Ergebnisterme der Regeln (11.10) und (11.11) unterscheiden sich von den entsprechenden Ergebnistermen der Regeln (11.1) und (11.2). Die äußeren `if`-Terme

$$\texttt{if}(a, \texttt{true}, \ldots) \quad \text{und} \quad \texttt{if}(a, \ldots, \texttt{false})$$

dokumentieren in der eigentlichen symbolischen Auswertung, dass das negierte Proof-Goal zur Auswertung der Subgoals verwendet werden darf. Da das Proof-Goal a in diesen `if`-Termen nicht ausgewertet werden soll, wird es mit der Annotation A_0 annotiert. Das Setzen der A-Umgebung für die Subgoals durch die Funktion $\mathbf{env}_{\mathcal{U}'}$

stellt sicher, dass die Subgoals in der eigentlichen symbolischen Auswertung genauso ausgewertet werden, wie in der Anwendungsbedingung (4) (vergleiche hierzu Abschnitt 8.9).

11.4 Auswahl der Klauseln und Pattern-Literale

Wir klären in diesem Abschnitt die Fragen, welche Klauseln und welche Instanzen der Klauseln zur symbolischen Auswertung verwendet werden sollen. Weiter besprechen wir, welche Literale einer Klausel als Pattern-Literale verwendet werden dürfen. Wir gehen hierzu wie folgt vor:

- In Abschnitt 11.4.1 erweitern wir die Menge der betrachteten Klauseln um die Resolventen aussagenlogischer Resolution. Durch diese Erweiterung ermöglichen wir die Verwendung von Quantifikationen, die ansonsten nur durch Erhöhung der maximalen Suchtiefe $depth_{\max}$ möglich wäre.
- In Abschnitt 11.4.2 beschreiben wir, wie die freien Variablen der Klauseln gezielt instantiiert werden können. Erst diese Instantiierungen ermöglichen Anwendungen von Quantifikationen wie der Transitivität während der symbolischen Auswertung.
- In Abschnitt 11.4.3 definieren wir schließlich welche Literale einer Klausel als Pattern-Literale verwendet werden dürfen. Diese Beschränkung der Pattern-Literale ist für die Effizienz der Regeln (11.10) und (11.11) enorm wichtig, da hierdurch der Suchraum deutlich reduziert wird.

Die in den Abschnitten 11.4.1 und 11.4.2 definierten Modifikationen vergrößern im Vergleich zu den Regeldefinitionen (11.10) und (11.11) den Suchraum der „*Assumption*“-Regeln. Dies ist notwendig, da ansonsten wichtige Anwendungen von Quantifikationen nicht möglich wären. Die in Abschnitt 11.4.3 definierte Modifikation reduziert den Suchraum dagegen.

11.4.1 Vervollständigung der Klauselmengen

Die mit einer Quantifikation q assoziierte Klauselmenge $\mathcal{C}_q$ ist unter Umständen für die symbolische Auswertung nicht optimal. Betrachten wir dazu ein Beispiel.

Beispiel 11.8. Sei die Multimenge $\mathcal{Q}$ durch $\{\!\{1 * q\}\!\}$ gegeben, wobei gilt

$$
\begin{array}{rcl}
q & = & \texttt{all t : tree, n : nat} \\
 & & \quad \texttt{if(completed(t),} \\
 & & \quad\quad \texttt{depth(insert(n, t))=succ(depth(t)),} \\
 & & \quad\quad \texttt{if(heap.structured(t), depth(insert(n, t))=depth(t), true))}\,.
\end{array}
$$

Die mit der Quantifikation q assoziierte Klauselmenge $\mathcal{C}_q$ ist dann gegeben durch

$$
\begin{array}{rcl}
C_1 & = & \{\neg\texttt{completed(t')}, \\
 & & \quad \texttt{depth(insert(n', l'))=succ(depth(t'))}\}, \\
C_2 & = & \{\texttt{completed(t')}, \\
 & & \quad \neg\texttt{heap.structured(t')}, \\
 & & \quad \texttt{depth(insert(n', t'))=depth(t')}\}\,.
\end{array}
$$

Betrachten wir nun die symbolische Auswertung des Terms

$$
\begin{array}{l}
\texttt{if(depth(insert(n, t))=succ(depth(t)),} \\
\quad \texttt{true,} \\
\quad \texttt{if(heap.structured(t),} \\
\quad\quad \texttt{depth(insert(n, t))=depth(t),} \\
\quad\quad \texttt{true))}
\end{array}
\tag{11.12}
$$

unter Verwendung der Klauselmenge $\{C_1, C_2\}$. Die Typoperatordefinition von `tree` und die Prozedurdefinitionen von `completed`, `depth` und `heap_structured` sind hierbei irrelevant. Wir gehen lediglich davon aus, dass es sich bei `completed`, `depth` und `heap_structured` um rekursiv-definierte Prozeduren handelt, deren Prozeduraufrufe in (11.12) nicht weiter ausgewertet werden können. Es ergibt sich dann mit Hilfe der vorläufigen Regeldefinitionen (11.10) und (11.11) die folgende symbolische Auswertung:[6]

$$
\begin{array}{ll}
\mathcal{U} \vdash & \texttt{if(depth(insert(n,t))=succ(depth(t)),} \\
& \quad \texttt{true,} \\
& \quad \texttt{if(heap.structured(t),} \qquad \overset{(C_2)}{\rightarrow} \\
& \quad\quad \underline{\texttt{depth(insert(n,t))=depth(t)}}\texttt{,} \\
& \quad\quad \texttt{true))} \\
& \texttt{if(depth(insert(n,t))=succ(depth(t)),} \\
& \quad \texttt{true,} \\
& \quad \texttt{if(heap.structured(t),} \qquad \overset{(C_1)}{\rightarrow} \\
& \quad\quad \texttt{if}(P_1, \texttt{true, if}(\underline{\neg \textit{completed}(\texttt{t})}, \textit{heap.structured}(\texttt{t}), \texttt{false})), \\
& \quad\quad \texttt{true))} \\
& \texttt{if(depth(insert(n,t))=succ(depth(t)),} \\
& \quad \texttt{true,} \\
& \quad \texttt{if(heap.structured(t),} \\
& \quad\quad \texttt{if}(P_1, \\
& \quad\quad\quad \texttt{true,} \qquad \rightarrow^* \\
& \quad\quad\quad \texttt{if}(\neg(\texttt{if}(P_2, \underline{\textit{depth}(\textit{insert}(\texttt{n},\texttt{t}))=\textit{succ}(\textit{depth}(\texttt{t}))}, \texttt{false})), \\
& \quad\quad\quad\quad \textit{heap.structured}(\texttt{t}), \\
& \quad\quad\quad\quad \texttt{false})), \\
& \quad\quad \texttt{true))} \\
& \quad \underline{\texttt{if(depth(insert(n,t))=succ(depth(t)),true,true)}} \rightarrow \\
& \quad \texttt{true.}
\end{array}
$$

Im zweiten Auswertungsschritt wurde zur Auswertung des Subgoals

$$\texttt{depth(insert(n,t))=depth(t)}$$

die Klausel C_1 verwendet. Das bedeutet, dass für die Auswertung des Terms (11.12) eine Suchtiefe von insgesamt 2 benötigt wird. Wir können diese Suchtiefe reduzieren, in dem wir bei symbolischer Auswertung von (11.12) zusätzlich die Klausel

$$
\begin{array}{rcl}
C_3 & = & \{\texttt{depth(insert(n',l'))=succ(depth(t'))}, \\
& & \quad \texttt{¬heap.structured(t')}, \\
& & \quad \texttt{depth(insert(n',t'))=depth(t')}\}
\end{array}
$$

verwenden. Bei dieser Klausel handelt es sich um eine Folgerung der Klauseln C_1 und C_2, d.h. es gilt

$$C_1 \wedge C_2 \Longrightarrow C_3.$$

Verwenden wir diese Klausel während der symbolischen Auswertung von (11.12) so ergibt sich die folgende Auswertung:[7]

[6] Mit P_1 und P_2 bezeichnen wir während der symbolischen Auswertung der Subgoals die Proof-Goals `depth(insert(n,t))=depth(t)` bzw. `completed(t)`.

[7] Mit P_1 bezeichnen wir während der symbolischen Auswertung der Subgoals wieder das Proof-Goal `depth(insert(n,t))=depth(t)`.

$$
\begin{array}{lll}
\mathcal{U} \vdash & \texttt{if(depth(insert(n,t))=succ(depth(t)),} & \\
& \quad \texttt{true,} & \\
& \quad \texttt{if(heap.structured(t),} & \overset{(C_3)}{\rightarrow} \\
& \quad\quad \underline{\texttt{depth(insert(n,t))=depth(t)}}\texttt{,} & \\
& \quad\quad \texttt{true))} & \\
& \texttt{if(depth(insert(n,t))=succ(depth(t)),} & \\
& \quad \texttt{true,} & \\
& \quad \texttt{if(heap.structured(t),} & \\
& \quad\quad \texttt{if(}P_1\texttt{,} & \\
& \quad\quad\quad \texttt{true,} & \rightarrow \\
& \quad\quad\quad \texttt{if(}\neg\texttt{(}\underline{\textit{depth(insert(n,t))=succ(depth(t))}}\texttt{),} & \\
& \quad\quad\quad\quad \textit{heap.structured(t)}\texttt{,} & \\
& \quad\quad\quad\quad \texttt{false)),} & \\
& \quad\quad \texttt{true))} & \\
& \qquad\qquad\qquad\qquad \vdots & \vdots \\
& \underline{\texttt{if(depth(insert(n,t))=succ(depth(t)),true,true)}} & \rightarrow \\
& \texttt{true} \ . &
\end{array}
$$

Im ersten Auswertungsschritt dieser Auswertung haben wir die Klauseln C_3 auf den Teilterm `depth(insert(n,t))=depth(t)` angewendet. Die dabei erzeugten Subgoals haben wir dann ohne Verwendung weiterer Klauseln zu `false` ausgewertet und so den Term insgesamt zu `true` vereinfacht. Die Suchtiefe dieser symbolischen Auswertung beträgt somit lediglich 1. □

Das Beispiel zeigt, dass wir die für eine erfolgreiche symbolische Auswertung benötigte Suchtiefe verringern können, wenn wir während der symbolischen Auswertung auch Klauseln betrachten, die lediglich Folgerungen der Klauseln aus $\mathcal{C}_q$ darstellen. Es ist daher sinnvoll die Klauselmengen $\mathcal{C}_q$ für die Regeln (11.10) und (11.11) um diese Klauseln zu erweitern. Die Klausel C_3 aus Beispiel 11.8 haben wir durch *aussagenlogische Resolution* aus den Klauseln C_1 und C_2 erhalten. Wir wollen daher für die Regeln (11.10) und (11.11) nicht allein die Klauselmengen $\mathcal{C}_q$ betrachten, sondern die mit Hilfe aussagenlogischer Resolution erweiterten Mengen. Zur Definition aussagenlogischer Resolution führen wir zunächst die Relation $\simeq$ ein. Diese Relation vergleicht Terme unter Berücksichtigung der Symmetrie der Gleichheit:

$$
\begin{array}{rcll}
x & \simeq & x & \text{gdw. } x \in \mathcal{V}(\Omega(P)), \\
f(t_1,\ldots,t_n) & \simeq & f(r_1,\ldots,r_n) & \text{gdw. } t_1 \simeq r_1,\ldots,t_n \simeq r_n, \\
t_1\texttt{=}t_2 & \simeq & r_1\texttt{=}r_2 & \text{gdw. } t_1 \simeq r_2, t_2 \simeq r_1,
\end{array}
$$

Aussagenlogische Resolution definieren wir dann wie folgt:

Definition 11.9 (Aussagenlogische Resolution)**.** Sei P ein Programm und l ein Literal über P. Weiter seien C, D Klauseln über P mit $l \in C$ und $\bar{l} \in_{\simeq} D$. Die Klausel

$$E = (C \setminus \{l\}) \cup (D \setminus_{\simeq} \{\bar{l}\})$$

heißt dann *aussagenlogischer Resolvent von C und D*. Die Klauseln C und D heißen *Elternklauseln des Resolventen E*. Die Menge aller aussagenlogischer Resolventen von C und D wird mit $\mathcal{Res}(C, D)$ bezeichnet. Für eine Menge $\mathcal{C}$ von Klauseln ist dann die Menge $\mathcal{Res}^*(\mathcal{C})$ definiert als kleinste Obermenge von $\mathcal{C}$ mit der Eigenschaft:

$$\forall\, C, D \in \mathcal{C} \text{ gilt } \mathcal{Res}(C, D) \subseteq \mathcal{Res}^*(\mathcal{C})\,.$$

□

Wir definieren dann die folgende Modifikation:

Modifikation (Auswahl der Klauseln)
Statt der Klauselmengen $\mathcal{C}_q$ werden in den Regeln (11.10) und (11.11) die Klauselmengen $\mathcal{R}es^*(\mathcal{C}_q)$ verwendet.

Anmerkung 11.10. Wir könnten statt der Menge $\mathcal{R}es^*(\mathcal{C}_q)$ auch die unter aussagenlogischer Resolution vervollständigte Menge $\mathcal{S}at^*(\mathcal{C}_q)$ betrachten. Da jedoch die Vervollständigung $\mathcal{S}at^*(\mathcal{C}_q)$ die Menge $\mathcal{C}_q$ insbesondere für Induktionshypothesen in einigen Fallstudien in unverhältnismäßiger Weise vergrößert, beschränken wir die Erweiterung der Menge $\mathcal{C}$ auf aussagenlogische Resolventen, für die beide Elternklauseln aus der Menge $\mathcal{C}_q$ stammen. □

Anmerkung 11.11. Zur Berechnung aussagenlogischer Resolventen berücksichtigen wir lediglich die Symmetrie der Gleichheit und nicht die aktuellen kommutativen Prozeduren. Andernfalls müssten nach Verifikation der Kommutativität eines Funktionssymbols die Klauselmengen neu berechnet werden, da sich unter Umständen andere Klauselmengen ergeben. □

11.4.2 Instantiierung der Subgoals

Die Anwendungsbedingung (2) der Regeln (11.10) und (11.11) stellt sicher, dass alle freien Typ- und Termvariablen der Klausel C durch den Matcher σ_ξ instantiiert werden. Diese Anwendungsbedingung schränkt jedoch die Verwendung der Regeln zu stark ein. Betrachten wir dazu ein Beispiel.

Beispiel 11.12. Sei $\mathcal{Q}$ die Multimenge $\{\!\{1 * q\}\!\}$, wobei q die folgende Quantifikation bezeichnet:

$$q \quad = \quad \texttt{all x : @a, y : @a, z : @a if(x=y, if(y=z, x=z, true), true)}\,.$$

Dann ist die mit q assoziierte Klauselmenge gegeben durch

$$\mathcal{C}_q \quad = \quad \{\underbrace{\{\neg(\texttt{x'=y'}), \neg(\texttt{y'=z'}), \texttt{x'=z'}\}}_{C_q}\}.$$

Wir können dann mit Hilfe von q und den Regeln (11.10) und (11.11) den Term

$$\texttt{if(x=y, if(y=z, x=z, true), true)} \tag{11.13}$$

nicht vereinfachen. Dies ist insbesondere deshalb unbefriedigend, da der Term mit dem Rumpf der Quantifikation q übereinstimmt und damit eine Auswertung zu `true` offensichtlich ist. Eine Anwendung der Klausel C_q mit Hilfe der Regel (11.10) bzw. (11.11) auf einen der Teilterme von (11.13) scheitert jedoch daran, dass durch die entsprechenden Matcher nicht alle freien Termvariablen instantiiert werden. Für den Teilterm `x=z` und das Literal `x'=z'` der Klausel C_q ergibt sich beispielsweise für die Regel (11.10) der Matcher σ_ϵ mit $\sigma = \{\texttt{x'/x}, \texttt{z'/z}\}$. Dieser Matcher instantiiert lediglich die Termvariablen `x'` und `z'` und nicht die Termvariable `y'`. Aufgrund der Anwendungsbedingung (2) können wir daher die Klausel C_q nicht auf den Teilterm `x=z` anwenden.

Ignorieren wir jedoch für die Anwendung der Regel (11.10) die Anwendungsbedingung (2) so erhalten wir für den Teilterm `x=z` aufgrund des erwähnten Matchers σ_ϵ die folgenden Subgoals:

$$d_1 = \neg(\texttt{x=y'}) \qquad\qquad d_2 = \neg(\texttt{y'=z})$$

Da die freie Termvariable $\mathtt{y}'$ die all-quantifizierte Termvariablen $\mathtt{y}$ der Quantifikation q repräsentiert, können wir $\mathtt{y}'$ in den Subgoals d_1 und d_2 durch einen beliebigen Term ersetzen. Insbesondere können wir $\mathtt{y}'$ durch $\mathtt{y}$ ersetzen, wodurch sich die folgenden Instanzen der Subgoals d_1 und d_2 ergeben:

$$\neg(\mathtt{x{=}y}) \qquad\qquad \neg(\mathtt{y{=}z})$$

Diese Terme können offensichtlich mit Hilfe der lokalen Hypothesen `x=y` und `y=z`, die während der Auswertung des Teilterms `x=z` in (11.13) zur Verfügung stehen, zu `false` ausgewertet werden. Das bedeutet, wir können auf den Teilterm `x=z` die Klausel C_q mit Hilfe der Regel (11.10) anwenden, sofern wir geeignete Instanzen der Subgoals d_1 und d_2 betrachten. Durch diese Anwendung der Klausel erhalten wir für (11.13) den folgenden Term:

$$\mathtt{if}(\mathtt{x{=}y}, \mathtt{if}(\mathtt{y{=}z}, \mathtt{if}(\mathtt{x{=}z}, \mathtt{true}, \mathtt{if}(\mathtt{x{=}y}, \mathtt{y{=}z}, \mathtt{false})), \mathtt{true}), \mathtt{true})$$

Diesen Term können wir offensichtlich zu `true` vereinfachen und erhalten somit die gewünschte Auswertung des Terms (11.13). □

Allgemein gilt, dass die Regeln (11.10) und (11.11) auch dann anwendbar sind, falls nicht alle Typ- und Termvariablen der Klauseln durch den Matcher instantiiert werden können, sofern für die Subgoals mit freien Typ- und Termvariablen geeignete Instanzen bestimmt und zu `false` ausgewertet werden können. Als besonders geeignete Instanzen der Subgoals sind hierbei Instanzen zu nennen, die sich aufgrund der aktuellen globalen und lokalen Hypothesenmenge ergeben. Betrachten wir die Subgoals d_1 und d_2 aus Beispiel 11.12 erneut, so erkennen wir, dass d_1 mit der Negation der lokalen Hypothesen `x=y` matcht und Subgoal d_2 mit der Negation der lokalen Hypothese `y=z`. In beiden Fällen ergibt sich als Matcher das Paar $\tilde{\sigma}_\epsilon$, wobei gilt $\tilde{\sigma} = \{\mathtt{y}'/\mathtt{y}\}$. Diesen Matcher haben wir in Beispiel 11.12 verwendet um die Instanzen der Subgoals zu bilden. Das besondere an diesen Instanzen ist, dass ihre Negationen in den lokalen Hypothesen enthalten sind und somit sichergestellt ist, dass sie auf sehr effiziente Weise zu `false` ausgewertet werden können. Allgemein betrachten wir daher ausschließlich solche Instanzen der Subgoals, die sich aufgrund negierter globaler oder lokaler Hypothesen bilden lassen. Zur Berechnung dieser Instanzen definieren wir die beiden folgenden Funktionen:

$$\begin{aligned} \mathit{inst\text{-}subst}_{\mathcal{U},H}(C) = \big\{ \varepsilon_{\simeq_\mathcal{U}}(\tilde{\sigma}_{\tilde{\xi}}) \;\big|\; & \tilde{\xi} \in \mathcal{Subst}_{tv'(C)}(\Omega(P)) \wedge \\ & \tilde{\sigma} \in \mathcal{Subst}_{fv'(\xi(C))}(\Omega(P), \Sigma(P)) \wedge \\ & \forall\, \nu' \in tv'(C)\ \exists l \in C \text{ mit } \nu' \in tv'(l) \wedge \overline{\tilde{\sigma}_{\tilde{\xi}}(l)} \in_{\simeq_\mathcal{U}} H \wedge \\ & \forall\, x' \in fv'(C)\ \exists l \in C \text{ mit } x' \in fv'(l) \wedge \overline{\tilde{\sigma}_{\tilde{\xi}}(l)} \in_{\simeq_\mathcal{U}} H \big\} \end{aligned}$$

$$\begin{aligned} \mathit{inst\text{-}triple}_{\mathcal{U},H}(a, C) = \big\{ \langle lit, \sigma_\xi, \tilde{\sigma}_{\tilde{\xi}} \rangle \;\big|\; & lit \in C \wedge \\ & \sigma_\xi \in match_{\simeq_\mathcal{U}}(lit, a) \wedge \\ & \tilde{\sigma}_{\tilde{\xi}} \in \mathit{inst\text{-}subst}_{\mathcal{U},H}(\sigma_\xi(C)) \big\} \end{aligned}$$

Die Funktionen haben die folgende Bedeutung:

- Die Funktion $\mathit{inst\text{-}subst}_{\mathcal{U},H}$ berechnet auf Basis der Hypothesen H für eine Klausel C die Menge aller Instantiierungen $\tilde{\sigma}_{\tilde{\xi}}$ der freien Variablen von C. Hierzu überprüfen die beiden ersten Bedingungen der Definition, dass ausschließlich freie Variablen durch die Substitutionen $\tilde{\xi}$ und $\tilde{\sigma}$ gebunden werden. Die beiden restlichen Bedingungen der Definition gewährleisten, dass jede freie Variable in C aufgrund eines Matches eines Literals $l \in C$ mit einer negierten Hypothese $h \in H$ instantiiert wird.

- Die Tripel $\langle lit, \sigma_\xi, \tilde{\sigma}_{\tilde{\xi}} \rangle$ der Menge $\mathit{inst\text{-}triple}_{\mathcal{U},H}$ repräsentieren für ein Proof-Goal a die für uns relevanten Instantiierungen der Klausel C. Hierbei bezeichnet lit das verwendete Pattern-Literal, σ_ξ den Matcher des Proof-Goals und des Pattern-Literals und $\tilde{\sigma}_{\tilde{\xi}}$ schließlich die Instantiierungen der freien Variablen von C, die nicht durch den Matcher festgelegt sind. Wir bezeichnen diese Tripel auch als *Instanz-Tripel*.

Die Berechnung der Instanz-Tripel $\mathit{inst\text{-}triple}_{\mathcal{U},H_\mathrm{L}\cup H_\mathrm{G}}(a, C)$ bzw. $\mathit{inst\text{-}triple}_{\mathcal{U},H_\mathrm{L}\cup H_\mathrm{G}}(\overline{a}, C)$ bzgl. der Hypothesenmenge $H_\mathrm{L} \cup H_\mathrm{G}$ ist nicht trivial und erfordert einen gewissen Berechnungsaufwand. Da die Anzahl der Instanz-Tripel, von der Anzahl der aktuellen lokalen und globalen Hypothesen abhängig ist, steigt der Aufwand zur Berechnung der Menge $\mathit{inst\text{-}triple}_{\mathcal{U},H_\mathrm{L}\cup H_\mathrm{G}}(a, C)$ je mehr lokale Hypothesen vorhanden sind. Um die Effizienz zu verbessern ist es daher sinnvoll nicht alle lokalen Hypothesen zur Berechnung der Instanz-Tripel zu verwenden. Wir verwenden zur Berechnung der Instanz-Tripel lediglich die lokalen Hypothesen, die durch `case`-Terme eingeführt werden, die bereits vor einer symbolischen Auswertung des Terms im Term enthalten waren. Das bedeutet insbesondere, dass wir lokale Hypothesen, die aufgrund von `case`-Termen erzeugt werden, die durch Auswertung von Prozeduraufrufen in den Term eingeführt wurden, nicht zur Berechnung der Instanz-Tripel heranziehen. Um dies zu erreichen, setzen wir vor der Auswertung eines Terms t das Instanz-Flag jedes `case`-Terms von t auf $\top$. Für `case`-Terme, die wir durch Auswertung eines Prozeduraufrufs in den Term einführen, setzen wir hingegen das Instanz-Flag auf $\bot$ (vergleiche hierzu die Definitionen der „*Execute*“- und „*Unfold*“-Regeln). Durch diese Annotationen stellen wir sicher, dass die Instanzhypothesenmenge H_I der aktuellen A-Umgebung genau die lokalen Hypothesen enthält, die aufgrund von `case`-Termen eingeführt werden, die bereits vor einer Auswertung des Terms im Term enthalten waren (vergleiche hierzu auch die Regel „*Evaluate branch*“). Wir verwenden somit zur Berechnung der Instanz-Tripel nicht mehr die lokale Hypothesenmenge H_L sondern die Instanzhypothesenmenge H_I.

Diese Überlegungen führen insgesamt zu folgender Modifikation:

Modifikation (Instantiierung der Klausel C):
Die Anwendungsbedingung (2) der Regeln (11.10) und (11.11) wird gestrichen. Die freien Variablen der Subgoals

$$\sigma_\xi(C) - \sigma_\xi(lit)$$

werden dann durch die Paare $\tilde{\sigma}_{\tilde{\xi}}$ der Instanztripel

$$\langle lit, \sigma_\xi, \tilde{\sigma}_{\tilde{\xi}} \rangle \in \mathit{inst\text{-}triple}_{\mathcal{U},H_\mathrm{I}\cup H_\mathrm{G}}(a, C)$$

gebunden.

11.4.3 „*Assumption*“-Heuristik

Bei den Regeln (11.10) und (11.11) darf jedes Literal einer Klausel als Pattern-Literal verwendet werden. Das bedeutet, dass für jedes Literal einer Klausel die Anwendung der Regeln (11.10) und (11.11) überprüft wird. Dies erzeugt einen entsprechend großen Suchraum. Symbolische Auswertungen auf Basis der Regeln (11.10) und (11.11) sind daher entsprechend ineffizient. Wir müssen somit die Verwendung der Literale der Klauseln als Pattern-Literal durch geeignete Heuristiken beschränken.

Betrachtet man die Literale einer Klausel genauer, so erkennt man, dass nicht alle Literale einer Klausel gleich gut als Pattern-Literal geeignet sind. Betrachten wir dazu beispielsweise die folgende Klausel:

$$\{\texttt{?empty}(\texttt{k}'), \texttt{member}(\texttt{minimum}(\texttt{k}'), \texttt{k}')\}\,. \tag{11.14}$$

Sofern nicht eine entsprechende globale oder lokale Hypothese zur Verfügung steht, ist es ausgesprochen unwahrscheinlich, das wir einen Term wie $\texttt{?empty}(t)$ aufgrund der Klausel (11.14) zu `true` auswerten können. Es ist daher nur bedingt sinnvoll das Literal $\texttt{?empty}(\texttt{k}')$ als Pattern-Literal zu verwenden. Wir teilen daher die Literale der Klauseln in geeignete und ungeeignete Pattern-Literale ein. Hierzu benötigen wir den Begriff des *linearen Atoms*:

Definition 11.13 (Lineare Atome)**.** Sei P ein Programm. Ein Atom über P der Form $f(t_1, \ldots, t_n)$ heißt *linear*, falls $t_1, \ldots, t_n$ paarweise verschiedene, freie Termvariablen bezeichnen. □

Definition 11.14 (Geeignete Pattern-Literale)**.** Sei P ein Programm, C eine Klausel über P und $lit \in C$. Das Literal lit heißt dann *geeignetes Pattern-Literal* von C, falls

(1) $|lit|$ ein nicht-lineares Atom ist, oder

(2) für alle $lit' \in C$ gilt $|lit'|$ ist linear.

Die Menge aller geeigneten Pattern-Literale einer Klausel C wird mit $\mathcal{Pat}(C)$ bezeichnet. □

Beispiel 11.15. Für die Klausel (11.14) ist $\texttt{member}(\texttt{minimum}(\texttt{k}'), \texttt{k}')$ ein geeignetes Pattern-Literal und $\texttt{?empty}(\texttt{k}')$ ein ungeeignetes Pattern-Literal. Für die Klausel der Transitivität der Gleichheit $\{\neg(\texttt{x}'\texttt{=}\texttt{y}'), \neg(\texttt{y}'\texttt{=}\texttt{z}'), \texttt{x}'\texttt{=}\texttt{z}'\}$ sind hingegen alle Literale der Klausel geeignete Pattern-Literale. □

Eine ausschließliche Verwendung von geeigneten Pattern-Literalen in den Regeln (11.10) und (11.11) stellt eine zu starke Einschränkung der Regeln dar. Einen Term wie

$$\texttt{if}(\texttt{member}(\texttt{minimum}(\texttt{k}), \texttt{k}), \texttt{true}, \texttt{?empty}(\texttt{k})) \tag{11.15}$$

könnten wir mit Hilfe der Regel (11.10) und der Klausel (11.14) dann nicht vereinfachen, obwohl wir aufgrund der lokalen Hypothese die Klausel in offensichtlicher Weise verwenden können. Wir benutzen daher in den Regeln (11.10) und (11.11) auch ungeeignete Pattern-Literale als Pattern-Literale, sofern ein Subgoal d allein mittels der lokalen oder globalen Hypothesen zu `false` ausgewertet werden kann. Aus Effizienzgründen beschränken wir uns hierbei jedoch auf lokale Hypothesen, die auch als Instanzhypothesen verwendet werden. Um zu überprüfen, ob wir ein Literal lit einer Klausel C für die Regeln (11.10) und (11.11) als Pattern-Literal verwenden, definieren wir daher das folgende Prädikat:

$$\mathcal{Pat}^{\sigma_\xi}_H(lit, C) \quad :\Longleftrightarrow \quad lit \in \mathcal{Pat}(C) \vee \sigma_\xi(C) \cap \overline{H} \neq \emptyset$$

Wir verwenden dann in den Regeln (11.10) und (11.11) das Literal $lit \in C$ als Pattern-Literal, falls gilt

$$\mathcal{Pat}^{\sigma_\xi}_{H_G \cup H_I}(lit, C).$$

Durch diese Festlegung können wir zwar den Term (11.15) mit Hilfe der Klausel (11.14) zu `true` auswerten, es existieren jedoch Fallstudien, in denen diese Beschränkung der Regeln immer noch zu stark ist. Dies führt dazu, dass wir die Regeln (11.10) und (11.11) auch dann anwenden, wenn

$$\mathcal{Top}_{H_I}(s, \tilde{\sigma}_{\tilde{\xi}}(\sigma_\xi(C)))$$

gilt, wobei das Prädikate $\mathcal{Top}_H(s, C)$ wie folgt definiert ist:

$$\mathcal{Top}_H(s, C) \quad :\Longleftrightarrow \quad s = 4^{depth_{\max}} \wedge C \cap \overline{H} \neq \emptyset$$

Das bedeutet, wir verwenden jedes Literal einer Klausel als Pattern-Literal, sofern wir nicht in der Auswertung eines Subgoals sind (Bedingung „$s = 4^{depth_{\max}}$“) und mindestens eines der erzeugten Subgoals durch die Instanzhypothesenmenge zu `false` ausgewertet werden kann (Bedingung „$C \cap \overline{H} \neq \emptyset$“). Weiter verwenden wir jedes Literal einer Klausel einer Induktionshypothese als Pattern-Literal. Wir definieren hierzu das folgende Prädikat:

$$\mathit{I\text{-}\mathcal{H}yp}(C) \quad :\Longleftrightarrow \quad C \text{ ist eine Klausel einer Induktionshypothese.}$$

Dies führt insgesamt zu der folgenden Modifikation:

Modifikation (Verwendung von Pattern-Literalen):
In den Regeln (11.10) und (11.11) werden ausschließlich Literale *lit* einer Klausel C als Pattern-Literal verwendet, für die die folgende Bedingung erfüllt ist:

$$\mathcal{Pat}^{\sigma_\xi}_{H_G \cup H_I}(lit, C) \vee \mathcal{Top}_{H_I}(s, \tilde{\sigma}_{\tilde{\xi}}(\sigma_\xi(C))) \vee \mathit{I\text{-}\mathcal{H}yp}_H(C).$$

Die Auswahl des Pattern-Literals auf Basis dieser Bedingung wird als „*Assumption Heuristic*“ bezeichnet.

11.5 Vollständige Regeldefinitionen

Wir fügen nun die in Abschnitt 11.4 definierten Anforderungen zusammen und definieren die endgültigen Auswertungsregeln:

61. Affirmative assumption

$$\frac{a^{\,A}}{\mathtt{if}^{\,A_{\mathrm{std}}}(\mathsf{a}_{A_0}(a), \mathtt{true}, \mathsf{env}_{\mathcal{U}'}(NOR(d_1,\ldots,d_n)))}, \qquad \begin{array}{|cc|}\hline & \text{S-Lim.} \\ \hline A\text{:} & s \\ \hline \end{array}$$

falls $s > 0$ und $\mathsf{e}(a) \in \mathcal{A}t(P)$ gilt. Weiter muss ein $C \in \bigcup_{q\in\mathcal{Q}} \mathcal{R}es^*(\mathcal{C}_q)$ existieren, so dass für ein $a' \in \mathit{struct\text{-}set}_P(\mathsf{e}(a))$ ein

$$\langle lit, \sigma_\xi, \tilde{\sigma}_{\tilde{\xi}}\rangle \in \mathit{INST} = \mathit{inst\text{-}triple}_{\mathcal{U},H_{\mathrm{I}}\cup H_{\mathrm{G}}}(a', C)$$

existiert, so dass die folgenden Bedingungen erfüllt sind:

(1) $\mathcal{P}at^{\sigma_\xi}_{H_{\mathrm{G}}\cup H_{\mathrm{I}}}(lit, C) \vee \mathcal{T}op_{H_{\mathrm{I}}}(s, \tilde{\sigma}_{\tilde{\xi}}(\sigma_\xi(C))) \vee \mathit{I\text{-}Hyp}_H(C)$.

(2) $s_{\mathrm{new}}(|\mathit{INST}|, |C|) \geq 1$.

(3) $\{d_1,\ldots,d_n\} = \mathsf{pol}_{\ominus}(\mathsf{a}_{A'}(\tilde{\sigma}_{\tilde{\xi}}(\sigma_\xi(C)) - \tilde{\sigma}_{\tilde{\xi}}(\sigma_\xi(lit))))$.

(4) $\forall i \in \{1,\ldots,n\}$ gilt $\mathcal{U}' \vdash d_i \rightarrow^! \mathtt{false}$.

Hierbei sind die A-Umgebung $\mathcal{U}'$ und die A-Annotation A' wie folgt gegeben:

	$\mathrm{Hyp}_{\mathrm{L}}$	$\mathrm{Hyp}_{\mathrm{I}}$	Quant.	Reg.
$\mathcal{U}'$:	$H_{\mathrm{G}} \cup \{\overline{\mathsf{e}(a)}\}$	H'_{I}	$\mathcal{Q} \setminus \{1 * q\}$	$\mathit{prim\text{-}execute}(\mathcal{R})$

A_{std}	S-Lim.
A':	$s_{\mathrm{new}}(\lvert \mathit{INST}\rvert, \lvert C\rvert)$

Die Menge H'_{I} ist wie folgt gegeben:

$$H'_{\mathrm{I}} = \begin{cases} H_{\mathrm{I}} \cup \{\overline{\mathsf{e}(a)}\} & \text{falls } s = 4^{\mathit{depth}_{\max}} \\ H_{\mathrm{I}} & \text{sonst .} \end{cases}$$

62. Negative assumption

$$\frac{a^{\,A}}{\mathtt{if}^{\,A_{\mathrm{std}}}(\mathsf{a}_{A_0}(a), \mathsf{env}_{\mathcal{U}'}(OR(d_1,\ldots,d_n), \mathtt{false}))}, \qquad \begin{array}{|cc|}\hline & \text{S-Lim.} \\ \hline A\text{:} & s \\ \hline \end{array}$$

falls $s > 0$ und $\mathsf{e}(a) \in \mathcal{A}t(P)$ gilt. Weiter muss ein $C \in \bigcup_{q\in\mathcal{Q}} \mathcal{R}es^*(\mathcal{C}_q)$ existieren, so dass für ein $a' \in \mathit{struct\text{-}set}_P(\mathsf{e}(a))$ ein

$$\langle lit, \sigma_\xi, \tilde{\sigma}_{\tilde{\xi}}\rangle \in \mathit{INST} = \mathit{inst\text{-}triple}_{\mathcal{U},H_{\mathrm{I}}\cup H_{\mathrm{G}}}(\overline{a'}, C)$$

existiert, so dass die folgenden Bedingungen erfüllt sind:

. . . die gleichen Bedingungen wie für Regel „Affirmative assumption“ . . .

Hierbei sind die A-Umgebung $\mathcal{U}'$ und die A-Annotation A' wie folgt gegeben:

	$\mathrm{Hyp}_{\mathrm{L}}$	$\mathrm{Hyp}_{\mathrm{I}}$	Quant.	Reg.
$\mathcal{U}'$:	$H_{\mathrm{G}} \cup \{\mathsf{e}(a)\}$	H'_{I}	$\mathcal{Q} \setminus \{1 * q\}$	$\mathit{prim\text{-}execute}(\mathcal{R})$

A_{std}	S-Lim.
A':	$s_{\mathrm{new}}(\lvert \mathit{INST}\rvert, \lvert C\rvert)$

Die Menge H'_{I} ist wie folgt gegeben:

$$H'_{\mathrm{I}} = \begin{cases} H_{\mathrm{I}} \cup \{\mathsf{e}(a)\} & \text{falls } s = 4^{\mathit{depth}_{\max}} \\ H_{\mathrm{I}} & \text{sonst .} \end{cases}$$

Anmerkung 11.16. Eine Anpassung der Search-Limits für die endgültigen Regeldefinitionen ist notwendig, um die Größe des Suchraums für eine Klausel C und ein Atom a auch in Abhängigkeit der Anzahl der verschiedenen Instanzen der Subgoals zu berechnen. Weiter möchten wir während der Auswertung der Subgoals das erste negierte Proof-Goal zur Instantiierung von Klauseln verwenden. Wir erweitern daher die Instanzhypothesenmenge H_I, falls wir uns nicht in der Auswertung eines Subgoals befinden. Die Beschränkung auf das *erste* negierte Proof-Goal zur Instantiierung ist aus Effizienzgründen notwendig. □

Für die Regeln „*Affirmative assumption*“ und „*Negative assumption*“ existiert der folgende Indeterminismus:

> Die Reihenfolge, in der die Quantifikationen und die Klauseln der Quantifikationen überprüft werden, wird durch die Regeln nicht festgelegt. Um diesen Indeterminismus zu beseitigen, ist im ✓eriFun-System mit der Multimenge $\mathcal{Q}$ eine Liste $L_\mathcal{Q}$ assoziiert, die alle Klauseln der vervollständigten Klauselmengen $\mathcal{Res}^*(C_q)$ enthält. [8] In dieser Liste sind die Klauseln – entsprechend ihrer Kardinalität – in aufsteigender Reihenfolge gespeichert. Die Regeln „*Affirmative assumption*“ und „*Negative assumption*“ überprüfen dann die Klauseln in der durch die Liste $L_\mathcal{Q}$ festgelegten Reihenfolge. Es werden somit zunächst die Klauseln mit weniger Literalen überprüft. Es ergeben sich dadurch üblicher Weise kürzere und übersichtlichere Auswertungen. Da durch die Kardinalität der Klauseln keine totale Ordnung auf den Klauseln definiert ist, lösen wir den verbleibenden Indeterminismus durch eine willkürliche Festlegung.

[8] Um auf effiziente Weise für ein Atom a die Klauseln C zur ermitteln, für die Instanz-Tripel existieren, ist es notwendig die Liste $L_\mathcal{Q}$ in Form eines Termindex zu speichern [22, 36]. Wir verwenden hierzu im ✓eriFun-System eine Variante eines Path-Index [43, 37, 32].

Kapitel 12

Die „*Replacement*“-Regel

Mit Hilfe von Quantifikationen bzw. der mit ihnen assoziierten Klauseln können wir nicht nur die Gültigkeit bzw. Ungültigkeit von Teiltermen des auszuwertenden Terms t nachweisen, sondern auch Termersetzungen durchführen.

Vorgehensweise (Termersetzung durch Quantifikationen):
Die Klauseln $C_i = \{l_1^i, \ldots, l_{n_i}^i\}$ der mit den Lemmata und Induktionshypothesen assoziierten Klauselmengen $\{C_1, \ldots, C_n\}$ können als Implikationen der Form

$$\overline{l_1^i} \wedge \ldots \wedge \overline{l_{j-1}^i} \wedge \overline{l_{j+1}^i} \wedge \ldots \wedge \overline{l_{n_i}^i} \Rightarrow l_j^i$$

aufgefasst werden, wobei l_j^i eine Gleichung der Form l=r bezeichnet. Um einen Term t mit $t \simeq_{\mathcal{U}} \sigma_\xi(l)$ durch $\sigma_\xi(r)$ ersetzen zu können, muss dann die Ungültigkeit der Literale $\sigma_\xi(l_k^i)$ mit $k \in \{1, \ldots, n\} \setminus \{j\}$ nachgewiesen werden. Hierzu werden die Literale symbolisch ausgewertet. Können alle Literale $\sigma_\xi(l_k^i)$ zu `false` ausgewertet werden, so kann der Term t durch $\sigma_\xi(r)$ ersetzt werden.

Die Literale $\sigma_\xi(l_k^i)$ mit $k \in \{1, \ldots, n\} \setminus \{j\}$ bezeichnen wir nachfolgend wieder als *Subgoals*. Zur Realisierung der Vorgehensweise definieren wir in diesem Kapitel die folgende Auswertungsregel:

$$\begin{array}{ll} \vdots & \vdots \\ 35. & \textit{Assumption replacement} \\ \vdots & \vdots \end{array}$$

Zur Definition der Regel gehen wir wie folgt vor. Für Termersetzungen ist eine der zentralen Fragen, wie Gleichungen der Form l=r gerichtet werden. Wir definieren daher in Abschnitt 12.4 die Relation $\succ_P$ zum Richten von Gleichungen. Wir ersetzen dann für eine Gleichung l=r einen Term $\sigma_\xi(l)$ durch $\sigma_\xi(r)$, falls $l \succ_P r$ gilt. Um die Definition der Relation $\succ_P$ in geeigneter Weise erklären zu können, ist es notwendig die Regel „*Assumption replacement*“ vor der Definition der Relation einzuführen. Wir geben daher in Abschnitt 12.1 zunächst eine vorläufige Definition der Auswertungsregel an. Diese Definition orientiert sich stark an der Definition der „*Assumption*“-Regeln aus Kapitel 11. Anschließend gehen wir in Abschnitt 12.2 auf die Wechselwirkungen mit anderen Regeln und auf die Annotationen des Terms $\sigma_\xi(r)$ ein, um dann in Abschnitt 12.3 die vollständige Regeldefinition angeben zu können.

12.1 Vorläufige Regeldefinition

Um Teiltermersetzungen auf Basis der in den Klauseln enthaltenen Gleichungen für die symbolische Auswertung zu ermöglichen, definieren wir die folgende vorläufige Regel.

$$\left.\begin{array}{l} \dfrac{t^{\,A}}{\mathtt{if}(\mathbf{env}_{\mathcal{U}'}(NOR(d_1,\ldots,d_n)),\mathbf{a}_{A_{\mathrm{std}}}(\sigma_\xi(r)),t^{\,A})}, \quad \dfrac{}{\begin{array}{c}\text{S-Lim.}\\ \hline A\colon\quad s\end{array}} \\[2ex] \text{falls } s>0 \text{ gilt und } \mathbf{e}(t) \text{ ein } \mathtt{case}\text{- und } \mathtt{let}\text{-freier Term ist. Weiter muss ein } C\in\bigcup_{q\in\mathcal{Q}}\mathcal{Res}^*(\mathcal{C}_q) \text{ und ein } lit\in C \text{ existieren mit} \\[1ex] \qquad\qquad \mathit{struct\text{-}eq}_P(lit)\simeq l{=}r \quad \text{und} \quad l\succ_P r, \\[1ex] \text{so dass ein } \sigma_\xi\in \mathit{match}_=(l,t) \text{ und ein } \tilde\sigma_{\tilde\xi}\in \mathit{inst\text{-}subst}(\sigma_\xi(C)) \text{ existieren, die die folgenden Bedingungen erfüllen:} \\[1ex] \quad (1)\ \{d_1,\ldots,d_n\}=\mathbf{a}_{A_{\mathrm{std}}}(\tilde\sigma_{\tilde\xi}(\sigma_\xi(C))-\sigma_\xi(lit)). \\[1ex] \quad (2)\ \forall i\in\{1,\ldots,n\} \text{ gilt } \mathcal{U}'\vdash d_i\to^!\mathtt{false}. \\[1ex] \text{Hierbei ist die A-Umgebung } \mathcal{U}' \text{ wie folgt gegeben:} \\[1ex] \qquad\qquad \dfrac{}{\begin{array}{c}\text{Reg.}\\ \hline \mathcal{U}'\colon\quad \mathit{no\text{-}execute}(\mathcal{R})\end{array}} \end{array}\right\} \quad (12.1)$$

Im Gegensatz zu den Regeln „*Affirmative assumption*“ und „*Negative assumption*“ wollen wir die Regel (12.1) (bzw. die endgültige Regeldefinition „*Assumption replacement*“) *vor* einer Auswertung der Argumente des Terms t anwenden. Dadurch stellen wir sicher, dass die Regel (12.1) auf alle Terme angewendet wird und nicht nur auf ausgewertete Terme. Die Anwendungsbedingungen der Regel werden daher deutlich häufiger überprüft, als die Anwendungsbedingungen der Regeln „*Affirmative assumption*“ und „*Negative assumption*“. Um eine entsprechend effizientere Überprüfung der Anwendungsbedingungen zu gewährleisten, haben wir daher in (12.1) die Verwendung von Quantifikationen und „*Functionality*“-Regeln sowie die Auswertung von Prozeduraufrufen während der symbolischen Auswertung der Subgoals d_i verboten. Weiter haben wir, um die Überprüfung der Regel für unnötig komplizierte Terme zu vermeiden, die Regel (12.1) auf `case`- und `let`-freie Terme t beschränkt. Die Berechnung des Matchers σ_ξ in (12.1) ohne Berücksichtigung der kommutativen Prozeduren ist hingegen notwendig um Endlos-Auswertungen für Prozeduren, die sowohl kommutativ als auch assoziativ sind, zu vermeiden (siehe hierzu beispielsweise [6] oder [7]). Betrachten wir zur Illustration der vorläufigen Regel ein Beispiel:

Beispiel 12.1. Sei P das Programm $\langle D_{\mathtt{list}}, D_{\mathtt{occurs}}, D_{\mathtt{insert}}, D_{\mathtt{isort}}\rangle$, wobei $D_{\mathtt{occurs}}$, $D_{\mathtt{insert}}$ und $D_{\mathtt{isort}}$ die Prozedurdefinitionen aus Abbildung 12.1 bezeichnen. Die Mengen der generalisierten Relationenbeschreibungen der Prozeduren sein gegeben durch

$$\mathit{grds}(\mathtt{occurs})=\mathit{grds}(\mathtt{insert})=\mathit{grds}(\mathtt{isort})=\{R\},$$

wobei gilt

$$R \;=\; \{\langle\langle \neg\mathtt{?empty(k)}\rangle,\{\mathtt{k/tl(k)}\}\rangle\}\,.$$

$$
\begin{array}{rcl}
D_{\texttt{occurs}} & = & \texttt{function occurs(n : nat, k : list[nat]) : nat} \Leftarrow \\
& & \quad \texttt{if(?empty(k),} \\
& & \quad\quad \texttt{0,} \\
& & \quad\quad \texttt{if(n=hd(k),} \\
& & \quad\quad\quad \texttt{succ(occurs(n, tl(k))),} \\
& & \quad\quad\quad \texttt{occurs(n, tl(k))))}
\end{array}
$$

$$
\begin{array}{rcl}
D_{\texttt{insert}} & = & \texttt{function insert(n : nat, k : list[nat]) : list[nat} \Leftarrow \\
& & \quad \texttt{if(?empty(k),} \\
& & \quad\quad \texttt{add(n, empty),} \\
& & \quad\quad \texttt{if(n>hd(k),} \\
& & \quad\quad\quad \texttt{add(hd(k), insert(n, tl(k))),} \\
& & \quad\quad\quad \texttt{add(n, k)))}
\end{array}
$$

$$
\begin{array}{rcl}
D_{\texttt{isort}} & = & \texttt{function isort(k : list[nat]) : list[nat} \Leftarrow \\
& & \quad \texttt{if(?empty(k),} \\
& & \quad\quad \texttt{empty,} \\
& & \quad\quad \texttt{insert(hd(k), isort(tl(k))))}
\end{array}
$$

Abbildung 12.1: Prozedurdefinitionen für `occurs`, `insert` und `isort`

Betrachten wir nun den Beweis des Lemmas

$$
\begin{array}{l}
\texttt{lemma isort\_permutes} \Leftarrow \texttt{all k : list[nat], n : nat} \\
\quad \texttt{occurs(n, k)=occurs(n, isort(k)),}
\end{array}
\tag{12.2}
$$

wobei wir die Gültigkeit der Lemmata

$$
\begin{array}{l}
\texttt{lemma occurs\_insert\_same*} \Leftarrow \texttt{all n, m : nat, k : list[nat]} \\
\quad \texttt{if(n=m, succ(occurs(n, k))=occurs(n, insert(m, k)), true),}
\end{array}
$$

$$
\begin{array}{l}
\texttt{lemma occurs\_insert\_different*} \Leftarrow \texttt{all k : list[nat], n, m : nat} \\
\quad \texttt{if(n=m, true, occurs(n, k)=occurs(n, insert(m, k)))}
\end{array}
$$

als gegeben voraussetzen. Zum Beweis des Lemmas (12.2) schlagen die Teilterme `occurs(n, k)` und `occurs(n, isort(k))` eine Induktion über die Relationenbeschreibung R vor. Für diese Induktion ergibt sich der folgende Induktionsschritt:

$$
\begin{array}{c}
\{\neg\texttt{?empty(k)}\}, \{\texttt{all n : nat occurs(n, tl(k))=occurs(n, isort(tl(k)))}\} \vdash \\
\texttt{occurs(n, k)=occurs(n, isort(k))}.
\end{array}
$$

Wir beweisen die Gültigkeit des Induktionsschritts, in dem wir den Goal-Term mit Hilfe der Multimenge $\mathcal{Q} = \{2 * lem_1, 2 * lem_2, 1 * ih\}$ und der globalen Hypothese $\neg$`?empty(k)` symbolische auswerten. Hierbei sind die Quantifikationen der Multi-

menge $\mathcal{Q}$ und die mit ihnen assoziierten Klauselmengen wie folgt gegeben:

$$\begin{aligned}
lem_1 &= \texttt{all n,m:nat,k:list[nat]} \\
&\quad \texttt{if(n=m,succ(occurs(n,k))=occurs(n,insert(m,k)),true)}, \\
lem_2 &= \texttt{all k:list[nat],n,m:nat} \\
&\quad \texttt{if(n=m,true,occurs(n,k)=occurs(n,insert(m,k)))}, \\
ih &= \texttt{all n:nat occurs(n,tl(k))=occurs(n,isort(tl(k)))},
\end{aligned}$$

$$\begin{aligned}
\mathcal{C}_{lem_1} &= \{\{\texttt{n'=m'},\texttt{succ(occurs(n',k'))=occurs(n',insert(m',k'))}\}\}, \\
\mathcal{C}_{lem_2} &= \{\{\neg(\texttt{n'=m'}),\texttt{occurs(n',k')=occurs(n',insert(m',k'))}\}\}, \\
\mathcal{C}_{ih} &= \{\{\texttt{occurs(n',tl(k))=occurs(n',isort(tl(k)))}\}\}.
\end{aligned}$$

Es ist zu beachten, dass die Gleichungen der Klauseln durch die Relationen $\succ_P$ wie folgt gerichtet werden:

$$\begin{aligned}
\texttt{occurs(n',insert(m',k'))} &\succ_P \texttt{succ(occurs(n',k'))}, \\
\texttt{occurs(n',insert(m',k'))} &\succ_P \texttt{occurs(n',k')}, \\
\texttt{occurs(n',isort(tl(k)))} &\succ_P \texttt{occurs(n',tl(k))}.
\end{aligned}$$

Die symbolische Auswertung des Goal-Terms des Induktionsschritts ist in Abbildung 12.2 dargestellt. Hierbei haben wir für jeden Auswertungsschritt, der die Regel (12.1) anwendet, die Klauselmenge angegeben, deren Klausel verwendet wird. Der Goal-Term kann offensichtlich mit Hilfe der Regel (12.1) zu `true` ausgewertet werden. Eine solche Auswertung ist ohne die Regel (12.1) nicht möglich. □

12.2 Anforderungen an Regelanwendungen

Die in Abschnitt 12.4 definierte Relation $\succ_P$ richtet Strukturgleichungen der Form

$$cons_i(sel_{i,1}(t),\ldots,sel_{i,n_i}(t))\texttt{=}t$$

mit $n_i > 0$ von links nach rechts. Das bedeutet, wir ersetzen mit Hilfe der Regel (12.1) einen Term der Form $cons_i(sel_{i,1}(t),\ldots,sel_{i,n_i}(t))$ durch t sofern eine Klausel C mit $\texttt{?}cons_i(t) \in C$ vorhanden ist und die entsprechenden Subgoals d_i zu `false` ausgewertet werden können. Solche Teiltermersetzungen können jedoch in Zusammenspiel mit den Regeln „*Replace left equality argument*“ und „*Replace right equality argument*“ zu Endlos-Auswertungen führen. Diese Regeln ersetzen nämlich einen Term t durch den Term $cons_i(sel_{i,1}(t),\ldots,sel_{i,n_i}(t))$, sofern ein entsprechender Strukturtest $\texttt{?}cons_i(t)$ in der lokalen oder globalen Hypothesenmenge enthalten ist. Um solche Endlos-Auswertungen zu verhindern, stellen wir die folgende Anforderung an die Anwendung der Regel (12.1):

Anforderungen (Strukturtests in Hypothesen)
Die Regel (12.1) darf Terme der Form

$$cons_i(sel_{i,1}(t),\ldots,sel_{i,n_i}(t))$$

nur dann durch den Term t ersetzen, falls der entsprechende Strukturtest $\texttt{?}cons_i(t)$ nicht in $H_\mathrm{G} \cup H_\mathrm{L}$ enthalten ist.

Alle Prozeduraufrufe im Term $\mathsf{a}_{A_{\mathrm{std}}}(\sigma_\xi(r))$ sind mit dem Label $\bot$ annotiert und können daher ausgewertet werden. Für Prozeduraufrufe, die bereits in t enthalten

$\mathcal{U} \vdash$ occurs(n, k)=occurs(n, isort(k))

$\rightarrow$ if(?empty(k), 0, if(n=hd(k), succ(occurs(n, tl(k))), occurs(n, tl(k))))=occurs(n, isort(k))

$\rightarrow$ if(false, 0, if(n=hd(k), succ(occurs(n, tl(k))), occurs(n, tl(k))))=occurs(n, isort(k))

$\rightarrow$ if(n=hd(k), succ(occurs(n, tl(k))), occurs(n, tl(k)))=occurs(n, isort(k))

$\rightarrow$ if(n=hd(k), succ(occurs(n, tl(k))), occurs(n, tl(k)))=occurs(n, if(?empty(k), empty, insert(hd(k), isort(tl(k)))))

$\rightarrow$ if(n=hd(k), succ(occurs(n, tl(k))), occurs(n, tl(k)))=occurs(n, if(false, empty, insert(hd(k), isort(tl(k)))))

$\rightarrow$ if(n=hd(k), succ(occurs(n, tl(k))), occurs(n, tl(k)))=occurs(n, insert(hd(k), isort(tl(k))))

$\rightarrow$ (mit $\mathcal{C}_{lem_1}$) if(n=hd(k), succ(occurs(n, tl(k))), occurs(n, tl(k)))=occurs(n, insert(hd(k), isort(tl(k)))))

$\rightarrow$ if(n=hd(k), succ(occurs(n, tl(k)))=if(n=hd(k), succ(occurs(n, isort(tl(k)))), ...), occurs(n, tl(k))=occurs(n, insert(hd(k), isort(tl(k)))))

$\rightarrow$ if(n=hd(k), succ(occurs(n, tl(k)))=if(true, succ(occurs(n, isort(tl(k)))), ...), occurs(n, tl(k))=occurs(n, insert(hd(k), isort(tl(k)))))

$\rightarrow$ (mit $\mathcal{C}_{ih}$) if(n=hd(k), succ(occurs(n, tl(k)))=succ(occurs(n, isort(tl(k)))), occurs(n, tl(k))=occurs(n, insert(hd(k), isort(tl(k)))))

$\rightarrow$ if(n=hd(k), succ(occurs(n, tl(k)))=succ(occurs(n, tl(k)), ...), occurs(n, tl(k))=occurs(n, insert(hd(k), isort(tl(k)))))

$\rightarrow$ if(n=hd(k), succ(occurs(n, tl(k)))=succ(occurs(n, tl(k))), occurs(n, tl(k))=occurs(n, insert(hd(k), isort(tl(k)))))

$\rightarrow$ (mit $\mathcal{C}_{lem_2}$) if(n=hd(k), true, occurs(n, tl(k))=occurs(n, insert(hd(k), isort(tl(k)))))

$\rightarrow$ if(n=hd(k), true, occurs(n, tl(k))=if(¬(n=hd(k)), occurs(n, isort(tl(k))), ...))

$\rightarrow$ if(n=hd(k), true, occurs(n, tl(k))=if(¬false, occurs(n, isort(tl(k))), ...))

$\rightarrow$ if(n=hd(k), true, occurs(n, tl(k))=if(true, occurs(n, isort(tl(k))), ...))

$\rightarrow$ (mit $\mathcal{C}_{ih}$) if(n=hd(k), true, occurs(n, tl(k))=occurs(n, isort(tl(k))))

$\rightarrow$ if(n=hd(k), true, occurs(n, tl(k))=if(true, occurs(n, tl(k)), ...))

$\rightarrow$ if(n=hd(k), true, occurs(n, tl(k))=occurs(n, tl(k)))

$\rightarrow$ if(n=hd(k), true, true)

$\rightarrow$ true

Abbildung 12.2: Symbolische Auswertung des Goal-Terms der Sequenz $\{\neg$?empty(l)$\}, \{$all n : nat occurs(n, tl(l))=occurs(n, isort(tl(l)))$\} \vdash$ occurs(n, l)=occurs(n, isort(l)).

sind, ist es jedoch sinnvoll die Labels aus t zu übernehmen. Wir formulieren daher die folgende Anforderung:

Anforderungen (Übernahme von Labels)
Bei der Ersetzung des Terms t durch den Term $\sigma_\xi(r)$ werden die Annotationen der gemeinsamen Teilterme aus t in den Term $\sigma_\xi(r)$ übernommen.

Wir ersetzen hierzu in $\mathsf{a}_{A_{std}}(\sigma_\xi(r))$ möglichst große Teilterme durch die entsprechenden Teilterme aus t. Hierzu definieren wir zunächst für annotierte Terme t und s die Menge aller Positionen von t für die sich $t|_\pi$ und s lediglich in den Annotationen unterscheiden:

$$\mathcal{P}os_s(t) \quad := \quad \{\pi \in \mathcal{P}os(t) \mid \mathsf{e}(s) = \mathsf{e}(t|_\pi)\}.$$

Diese Menge verwenden wir um die Annotationen der größten gemeinsamen Teilterme zweier annotierter Terme t und r von t nach r zu übertragen. Wir definieren hierzu die Funktion $\mathsf{rep}_t(r)$:[1]

$$\begin{array}{rcll} \mathsf{rep}_t(x) & := & x, & \\ \mathsf{rep}_t(f^{\,A}(t_1,\ldots,t_n)) & := & f^{\,A}(\mathsf{rep}_t(t_1),\ldots,\mathsf{rep}_t(t_n)) & \text{falls } \mathcal{P}os_{f^{\,A}(\ldots)}(t) = \emptyset, \\ \mathsf{rep}_t(f^{\,A}(t_1,\ldots,t_n)) & := & t|_{min_{>_{\mathcal{P}os}}(\mathcal{P}os_{f^{\,A}(\ldots)}(t))} & \text{falls } \mathcal{P}os_{f^{\,A}(\ldots)}(t) \neq \emptyset. \end{array}$$

12.3 Vollständige Regeldefinition

Unter Berücksichtigung der im vorherigen Abschnitt definierten Anforderungen definieren wir die Regel „*Assumption replacement*“ wie folgt:

[1] Die Verwendung der $>_{\mathcal{P}os}$-minimalen Position aus $\mathcal{P}os_s(t)$ ist notwendig, um die Funktion rep_t eindeutig zu definieren. Jede andere Position in $\mathcal{P}os_s(t)$ ist prinzipiell genauso geeignet. Man muss sich zur Definition von rep_t aber auf eine Position festlegen.

35. Assumption replacement

$$\frac{t^{\boxed{A}}}{\texttt{if}(\mathbf{env}_{\mathcal{U}'}(NOR(d_1,\ldots,d_n)),\mathbf{rep}_t(\mathbf{a}_{A_{\text{std}}}(\sigma_\xi(r))),t)},\qquad \frac{}{\begin{array}{c|c}\multicolumn{2}{c}{\text{S-Lim.}}\\ \hline A\text{:} & s\end{array}}$$

falls $s > 0$ gilt und $\mathbf{e}(a)$ ein `case`- und `let`-frei Term ist. Weiter muss ein $C \in \bigcup_{q\in\mathcal{Q}} \mathcal{Res}^*(\mathcal{C}_q)$ und ein $lit \in C$ existieren mit

$$\mathit{struct\text{-}eq}_P(lit) \simeq l{=}r \quad \text{und} \quad l \succ_P r,$$

so dass ein $\sigma_\xi \in \mathit{match}_=(l,t)$ und ein $\tilde{\sigma}_{\tilde{\xi}} \in \mathit{inst\text{-}subst}(\sigma_\xi(C))$ existieren, die die folgenden Bedingungen erfüllen:

(1) $\{d_1,\ldots,d_n\} = \mathbf{a}_{A_{\text{std}}}(\tilde{\sigma}_{\tilde{\xi}}(\sigma_\xi(C)) - \tilde{\sigma}_{\tilde{\xi}}(\sigma_\xi(lit)))$.

(2) $\forall i \in \{1,\ldots,n\}$ gilt $\mathcal{U}' \vdash d_i \to^! \texttt{false}$.

(3) $lit = \texttt{?}\mathit{cons}(t) \Longrightarrow \texttt{?}\mathit{cons}(t) \notin_{\simeq_\mathcal{U}} H_\text{G} \cup H_\text{L}$.

(4) $\overline{\sigma_\xi(lit)} \notin H_\text{G} \cup H_\text{L}$.

Hierbei ist die A-Umgebung $\mathcal{U}'$ wie folgt gegeben:

	Reg.
$\mathcal{U}'$:	$\mathit{no\text{-}execute}(\mathcal{R})$

Die Anwendungsbedingung (4) der Regel „*Assumption replacement*“ verhindert die Verwendung von Gleichungen, die aufgrund der lokalen und globalen Hypothesenmengen ungültig sind. Die Verwendung dieser ungültigen Gleichungen führte in älteren Versionen des Auswertungskalküls in manchen Fallstudien zu Endlos-Auswertungen. Diese Endlos-Auswertungen sind jedoch mit unserer aktuellen Definition des Auswertungskalküls nicht mehr reproduzierbar. Da die Verwendung ungültiger Gleichungen jedoch prinzipiell nicht sinnvoll ist, ist die Anwendungsbedingung auch ohne Endlos-Auswertungen eine sinnvolle Beschränkung der Regel.

12.4 Relation zum Richten von Gleichungen

Wir führen in diesem Abschnitt die Relation $\succ_P$ zum Richten von Gleichungen der Form $l{=}r$ ein. Die Definition der Relation orientiert sich hierbei an den lexikographischen Pfad-Ordnungen aus dem Gebiet der Termersetzungssysteme [7, 15]. Wir geben zunächst in Abschnitt 12.4.1 die vollständige Definition der Relation an und illustrieren ihre Funktionsweise und Nützlichkeit anschließend in Abschnitt 12.4.2 anhand von Beispielen.

12.4.1 Definition der Relation

Um die Relation $\succ_P$ formal definieren zu können, benötigen wir zunächst zwei weitere Relationen: $\succ_{\text{uses},P}$ und $\succsim_{\text{emb},P}$.

Definition 12.2 ($\succ_{\text{uses},P}$)**.** Sei P ein Programm. Die Relation $\succ_{\text{uses},P} \subseteq \mathit{dom}(\Sigma(P)) \times \mathit{dom}(\Sigma(P))$ ist definiert als:

$$f \succ_{\text{uses},P} g$$

gdw. f eine Prozedur bezeichnet, $f \neq g$ gilt und g im Prozedurrumpf von f verwendet wird. □

Für zwei Funktionssymbole f und g gilt somit $f \succ^{+}_{\text{uses},P} g$ gdw. f eine Prozedur ist und das Funktionssymbol g direkt oder indirekt zur Definition von f benötigt wird. Betrachten wir dazu ein Beispiel.

Beispiel 12.3. Für das Programm $P = \langle D_{\texttt{plus}}, D_{\texttt{dbl}}, D_{\texttt{half}}, D_{\texttt{even}}, D_{\texttt{mult}} \rangle$ mit den entsprechenden Prozedurdefinitionen aus Abbildung 8.2 gilt

$$\texttt{mult} \succ^{+}_{\text{uses},P} \texttt{pred},$$

da sowohl $\texttt{mult} \succ_{\text{uses},P} \texttt{plus}$ als auch $\texttt{plus} \succ_{\text{uses},P} \texttt{pred}$ gilt. □

Definition 12.4 ($\gtrsim_{\text{emb},P}$)**.** Sei P ein Programm. Die Relation $\gtrsim_{\text{emb},P} \subseteq \mathcal{T}erm(P) \times \mathcal{T}erm(P)$ ist definiert als:

$$r \gtrsim_{\text{emb},P} l$$

gdw. r einen Teilterm der Form $f(s^*)$ enthält und dieser Teilterm die folgenden Bedingungen erfüllt:

(1) $f \in \Sigma^{\text{proc}}(P)$, und

(2) $\exists \pi \in \mathcal{P}os(|R_f[x^*/s^*]|_{\mathcal{P}})$ mit $match_{=}(l, |R_f[x_*/s^*]|_{\mathcal{P}}\big|_\pi) \neq \emptyset$.

□

Für zwei Terme l und r gilt somit $r \gtrsim_{\text{emb},P} l$, falls durch Auswertung eines Prozeduraufrufs in r eine Instanz des Terms l erzeugt werden kann:

$$\begin{array}{rcl} r & = & \ldots f(s^*) \ldots \\ & & \downarrow \\ & & R_f[x^*/s^*] \quad \geq_{\mathcal{T}erm} \quad \sigma(l) \end{array}$$

Betrachten wir dazu ein Beispiel:

Beispiel 12.5. Für das Programm $P = \langle D_{\texttt{plus}}, D_{\texttt{dbl}}, D_{\texttt{half}}, D_{\texttt{even}}, D_{\texttt{mult}} \rangle$ mit den entsprechenden Prozedurdefinitionen aus Abbildung 8.2 gilt

$$\texttt{mult}(\texttt{x}', \texttt{y}') \gtrsim_{\text{emb},P} \texttt{plus}(\texttt{x}', \texttt{y}'),$$

da der instantiierte Prozedurrumpf von `mult` den Teilterm

$$\texttt{plus}(\texttt{mult}(\texttt{half}(\texttt{x}'), \texttt{dbl}(\texttt{y}')), \texttt{y}')$$

enthält und dieser Teilterm offensichtlich eine Instanz von $\texttt{plus}(\texttt{x}', \texttt{y}')$ ist. □

Kommen wir nun zur Definition der Relation $\succ_P$.

Definition 12.6 ($\succ_P$)**.** Sei P ein Programm. Die Relation $\gtrsim_P$ ist dann auf `case`- und `let`-freien Termen l und r wie folgt definiert:

$$l \gtrsim_P r$$

gdw. $l = r$ oder die folgenden Bedingungen erfüllt sind:

(1) $tv'(l) \supseteq tv'(r)$,

(2) $fv'(l) \supseteq fv'(r)$,

(3) $r \not\geq_{\mathcal{T}erm} l$,

(4) $r \not\gtrsim_{\text{emb},P} l$,

(5) $l \gtrsim'_P r$.

Die Relation $\succsim'_P$ ist hierbei wie folgt definiert:

$$l \succsim'_P r$$

gdw. entweder

(1) $l = f(l_1, \ldots, l_n)$ und r eine Termvariable mit $l \geq_{\mathcal{T}erm} r$ ist, oder

(2) $l = f(l_1, \ldots, l_n)$, $r = g(r_1, \ldots, r_m)$ und eine der folgende Bedingungen erfüllt ist:

(a) $f \in \Sigma(P) \setminus \Sigma^{\text{cons}}(P)$ und $g \in \Sigma^{\text{cons}}(P)$.

(b) $(f \in \Sigma^{\text{cons}}(P) \Longrightarrow l \geq_{\mathcal{T}erm} r)$, $g \not\succ^+_{\text{uses},P} f$ und $\exists i \in \{1, \ldots, n\}$ mit $l_i \succsim_P r$.

(c) $f, g \notin \Sigma^{\text{cons}}(P)$, $\forall i \in \{1, \ldots, m\}$ gilt $l \succsim_P r_i$, $\exists j \in \{1, \ldots, m\}$ mit $l \succ_P r_j$ und eine der folgenden Bedingungen erfüllt ist:

(i) $(g \in \Sigma^{\text{proc}}(P) \Longrightarrow f \in \Sigma^{\text{proc}}(P))$ und g ist nicht in l enthalten.
(ii) $g \neq f$, $f \in \Sigma^{\text{proc}}(P)$ und $f \succ^+_{\text{uses},P} g$.
(iii) $n < m$.

(d) $f = g$ und $\exists i \in \{1, \ldots, n\}$ mit $l_i \succ_P r_i$ und $\forall j \in \{1, \ldots, i-1\}$ gilt $l_i \approx_P r_i$.

Wir schreiben $l \succ_P r$ falls $l \succsim_P r$ und $r \not\succsim_P l$ gilt. Weiter schreiben wir $l \approx_P r$ falls $l \succsim_P r$ und $r \succsim_P r$ gilt. □

Da der Aufwand zum Richten der Gleichungen durch die Relation $\succ_P$ nicht trivial ist, wird eine Gleichung l=r in ✓eriFun nur dann gerichtet, wenn die Gleichung die folgende Bedingung erfüllt:

$$\#(l) * \#(r) \leq \textit{max-size}\,.$$

Hierbei bezeichnet $\#(t)$ die Anzahl der Funktionssymbole und Termvariablen in t und *max-size* eine vom ✓eriFun-System vorgegebene Konstante. In der Version 3.0 des Systems gilt *max-size* $= 100$.

12.4.2 Beispiele

Wir betrachten nun einige Beispiele, um die Nützlichkeit der Relation $\succ_P$ zum Richten von Gleichungen zu illustrieren und um die Funktionsweise der Relation zu erläutern.

Beispiel 12.7. Sei P das Programm $\langle D_{\texttt{plus}} \rangle$, wobei $D_{\texttt{plus}}$ die entsprechende Prozedurdefinition aus Abbildung 8.2 bezeichnet. Für die Gleichung

$$\texttt{plus(x', succ(y'))=succ(plus(x', y'))}$$

gilt aufgrund der Bedingung (2)(a) der Definition der Relation $\succsim'_P$

$$\texttt{plus(x', succ(y'))} \succsim_P \texttt{succ(plus(x', y'))}.$$

Weiter gilt $\texttt{succ(plus(x', y'))} \not\succsim_P \texttt{plus(x', succ(y'))}$ und damit

$$\texttt{plus(x', succ(y'))} \succ_P \texttt{succ(plus(x', y'))}.$$

Das bedeutet, wir ersetzen durch die Auswertungsregel „*Assumption replacement*" jede Instanz von $\mathtt{plus}(\mathtt{x}', \mathtt{succ}(\mathtt{y}'))$ durch die entsprechende Instanz von $\mathtt{succ}(\mathtt{plus}(\mathtt{x}', \mathtt{y}'))$. Dies ist sinnvoll, da wir so den Konstruktor aus dem plus-Term herausziehen und so zusätzliche Auswertungen ermöglichen. Weiter ist die so gerichtete Gleichung „kompatibel" mit der Auswertung der Prozeduraufrufe von plus durch die Regel „*Execute procedure call (recursive cases)**". □

Beispiel 12.8. Sei P das Programm $\langle D_{\mathtt{list}}, D_{\mathtt{occurs}}, D_{\mathtt{insert}}\rangle$, wobei $D_{\mathtt{list}}$ die entsprechende Typoperatordefinition aus Abbildung 3.1 und $D_{\mathtt{occurs}}$ sowie $D_{\mathtt{insert}}$ die entsprechenden Prozedurdefinitionen aus Abbildung 12.1 bezeichnen. Es gelten dann die folgenden Orientierungen:[2]

$$\begin{array}{rcl} \mathtt{insert}(\mathtt{m}', \mathtt{k}') & \succ_P & \mathtt{k}' \\ \mathtt{occurs}(\mathtt{n}', \mathtt{insert}(\mathtt{m}', \mathtt{k}')) & \succsim_P & \mathtt{occurs}(\mathtt{n}', \mathtt{k}'). \end{array} \quad \stackrel{(2)(d)}{\Longrightarrow}$$

Da außerdem $\mathtt{occurs}(\mathtt{n}', \mathtt{k}') \not\succsim_P \mathtt{occurs}(\mathtt{n}', \mathtt{insert}(\mathtt{m}', \mathtt{k}'))$ gilt, folgt

$$\mathtt{occurs}(\mathtt{n}', \mathtt{insert}(\mathtt{m}', \mathtt{k}')) \succ_P \mathtt{occurs}(\mathtt{n}', \mathtt{k}').$$

Mit Hilfe des Lemmas

$$\begin{array}{l} \mathtt{lemma\ occurs\_diff} \Leftarrow \mathtt{all\ k : list[nat], n, m : nat} \\ \quad \mathtt{if(n{=}m, true, occurs(n, insert(m, k)){=}occurs(n, k))} \end{array}$$

können wir dann Prozeduraufrufe von insert in occurs-Termen entfernen und so occurs-Terme deutlich vereinfachen. □

Beispiel 12.9. Sei P das Programm $\langle D_{\mathtt{list}}, D_{\mathtt{plus}}, D_{\mathtt{occurs}}, D_{\mathtt{dis\_ev}}, D_{\mathtt{dis\_odd}}, D_{\mathtt{merge}}\rangle$, wobei $D_{\mathtt{list}}$ die entsprechende Typoperatordefinition aus Abbildung 3.1 und $D_{\mathtt{plus}}$, $D_{\mathtt{occurs}}$, $D_{\mathtt{dis\_ev}}$, $D_{\mathtt{dis\_odd}}$ sowie $D_{\mathtt{merge}}$ die entsprechenden Prozedurdefinitionen aus den Abbildungen 8.2, 12.1 und 11.1 bezeichnen. Es gelten dann die folgenden Orientierungen:

$$\begin{array}{rcl} \mathtt{merge}(\mathtt{k}', \mathtt{l}') & \succ_P & \mathtt{k}' \\ \mathtt{occurs}(\mathtt{n}', \mathtt{merge}(\mathtt{k}', \mathtt{l}')) & \succsim_P & \mathtt{occurs}(\mathtt{n}', \mathtt{k}') \end{array} \quad \stackrel{(2)(d)}{\Longrightarrow}$$

$$\begin{array}{rcl} \mathtt{merge}(\mathtt{k}', \mathtt{l}') & \succ_P & \mathtt{l}' \\ \mathtt{occurs}(\mathtt{n}', \mathtt{merge}(\mathtt{k}', \mathtt{l}')) & \succsim_P & \mathtt{occurs}(\mathtt{n}', \mathtt{l}'). \end{array} \quad \stackrel{(2)(d)}{\Longrightarrow}$$

Da außerdem $\mathtt{occurs}(\mathtt{n}', \mathtt{k}') \not\succsim_P \mathtt{occurs}(\mathtt{n}', \mathtt{merge}(\mathtt{k}', \mathtt{l}'))$ gilt, folgt aus Bedingung (2)(c)(i) der Definition von $\succsim'_P$

$$\mathtt{occurs}(\mathtt{n}', \mathtt{merge}(\mathtt{k}', \mathtt{l}')) \succsim_P \mathtt{plus}(\mathtt{occurs}(\mathtt{n}', \mathtt{k}'), \mathtt{occurs}(\mathtt{n}', \mathtt{l}')).$$

Mit $\mathtt{plus}(\mathtt{occurs}(\mathtt{n}', \mathtt{k}'), \mathtt{occurs}(\mathtt{n}', \mathtt{l}')) \not\succsim_P \mathtt{occurs}(\mathtt{n}', \mathtt{merge}(\mathtt{k}', \mathtt{l}'))$ folgt dann

$$\mathtt{occurs}(\mathtt{n}', \mathtt{merge}(\mathtt{k}', \mathtt{l}')) \succ_P \mathtt{plus}(\mathtt{occurs}(\mathtt{n}', \mathtt{k}'), \mathtt{occurs}(\mathtt{n}', \mathtt{l}')).$$

Wir können somit mit Hilfe des Lemmas

$$\begin{array}{l} \mathtt{lemma\ occurs\_merge} \Leftarrow \mathtt{all\ k, l : list[nat], n : nat} \\ \quad \mathtt{occurs(n, merge(k, l)){=}plus(occurs(n, k), occurs(n, l))} \end{array}$$

die Prozedur merge aus occurs-Termen herausziehen. Die sich dadurch ergebende Summe $\mathtt{plus}(\mathtt{occurs}(\mathtt{n}, \mathtt{k}), \mathtt{occurs}(\mathtt{n}, \mathtt{l}))$ ist für die symbolische Auswertung besser geeignet, als der ursprüngliche Term, da die Prozedurdefinition von plus einfacher ist als die Prozedurdefinition von merge. □

[2] Wir haben die Bedingungen, die für die Orientierung ursächlich sind, oberhalb der Implikationspfeile notiert. Die erste Orientierung ist aufgrund der Bedingung (1) der Definition von $\succsim'_P$ gültig.

Beispiel 12.10. Sei P das Programm $\langle D_{\mathtt{plus}}, D_{\mathtt{times}}\rangle$, wobei $D_{\mathtt{plus}}$ und $D_{\mathtt{times}}$ die entsprechenden Prozedurdefinitionen aus Abbildung 8.2 und Beispiel 10.3 bezeichnen. Es gelten dann die folgenden Orientierungen:

$$\begin{array}{rcl} \mathtt{plus}(\mathtt{y}',\mathtt{z}') & \succ_P & \mathtt{y}' \\ \mathtt{times}(\mathtt{x}',\mathtt{plus}(\mathtt{y}',\mathtt{z}')) & \succsim_P & \mathtt{times}(\mathtt{x}',\mathtt{y}') \end{array} \overset{(2)(d)}{\Longrightarrow}$$

$$\begin{array}{rcl} \mathtt{plus}(\mathtt{y}',\mathtt{z}') & \succ_P & \mathtt{z}' \\ \mathtt{times}(\mathtt{x}',\mathtt{plus}(\mathtt{y}',\mathtt{z}')) & \succsim_P & \mathtt{times}(\mathtt{x}',\mathtt{z}'). \end{array} \overset{(2)(d)}{\Longrightarrow}$$

Da außerdem $\mathtt{times}(\mathtt{x}',\mathtt{y}') \not\succsim_P \mathtt{times}(\mathtt{x}',\mathtt{plus}(\mathtt{y}',\mathtt{z}'))$ gilt, folgt aus Bedingung (2)(c)(ii) der Definition von $\succsim'_P$

$$\mathtt{times}(\mathtt{x}',\mathtt{plus}(\mathtt{y}',\mathtt{z}')) \succsim_P \mathtt{plus}(\mathtt{times}(\mathtt{x}',\mathtt{y}'),\mathtt{times}(\mathtt{x}',\mathtt{z}')).$$

Mit $\mathtt{plus}(\mathtt{times}(\mathtt{x}',\mathtt{y}'),\mathtt{times}(\mathtt{x}',\mathtt{z}')) \not\succsim_P \mathtt{times}(\mathtt{x}',\mathtt{plus}(\mathtt{y}',\mathtt{z}'))$ gilt dann

$$\mathtt{times}(\mathtt{x}',\mathtt{plus}(\mathtt{y}',\mathtt{z}')) \succ_P \mathtt{plus}(\mathtt{times}(\mathtt{x}',\mathtt{y}'),\mathtt{times}(\mathtt{x}',\mathtt{z}')).$$

Wir multiplizieren dann mit Hilfe des Lemmas

$$\begin{array}{l} \mathtt{lemma\ times\_distribute} \Leftarrow \mathtt{all\ x,y,z : nat} \\ \quad \mathtt{times(x,plus(y,z))=plus(times(x,y),times(x,z))} \end{array}$$

und der Regel „*Assumption replacement*“ entsprechende `times-plus`-Terme aus. Dieses Ausmultiplizieren führt zu einer Normalisierung arithmetischer Ausdrücke. □

Beispiel 12.11. Sei P das Programm $\langle D_{\mathtt{plus}}, D_{\mathtt{dbl}}\rangle$, wobei $D_{\mathtt{plus}}$ und $D_{\mathtt{dbl}}$ die entsprechenden Prozedurdefinitionen aus den Abbildungen 8.2 bezeichnen. Es gelten dann die folgenden Orientierungen:

$$\begin{array}{rcl} \mathtt{plus}(\mathtt{x}',\mathtt{y}') & \succ_P & \mathtt{x}' \\ \mathtt{dbl}(\mathtt{plus}(\mathtt{x}',\mathtt{y}')) & \succsim_P & \mathtt{dbl}(\mathtt{x}') \end{array} \overset{(2)(d)}{\Longrightarrow}$$

$$\begin{array}{rcl} \mathtt{plus}(\mathtt{x}',\mathtt{y}') & \succ_P & \mathtt{y}' \\ \mathtt{dbl}(\mathtt{plus}(\mathtt{x}',\mathtt{y}')) & \succsim_P & \mathtt{dbl}(\mathtt{y}'). \end{array} \overset{(2)(d)}{\Longrightarrow}$$

Da außerdem $\mathtt{dbl}(\mathtt{x}') \not\succsim_P \mathtt{dbl}(\mathtt{plus}(\mathtt{x}',\mathtt{y}'))$ gilt, folgt aus Bedingung (2)(c)(iii) der Definition von $\succsim'_P$

$$\mathtt{dbl}(\mathtt{plus}(\mathtt{x}',\mathtt{y}')) \succsim_P \mathtt{plus}(\mathtt{dbl}(\mathtt{x}'),\mathtt{dbl}(\mathtt{y}')).$$

Mit $\mathtt{plus}(\mathtt{dbl}(\mathtt{x}'),\mathtt{dbl}(\mathtt{y}')) \succ_P \mathtt{dbl}(\mathtt{plus}(\mathtt{x}',\mathtt{y}'))$ folgt dann

$$\mathtt{dbl}(\mathtt{plus}(\mathtt{x}',\mathtt{y}')) \succ_P \mathtt{plus}(\mathtt{dbl}(\mathtt{x}'),\mathtt{dbl}(\mathtt{y}')).$$

Wir ziehen somit mit Hilfe des Lemmas

$$\begin{array}{l} \mathtt{lemma\ dbl\_plus} \Leftarrow \mathtt{all\ x,y : nat} \\ \quad \mathtt{dbl(plus(x,y))=plus(dbl(x),dbl(y))} \end{array}$$

die Prozedur `dbl` in `plus`-Terme hinein. Das bedeutet, dass wir auch solche Terme durch Anwendung der Regel „*Assumption replacement*“ ausmultiplizieren. □

$D_{\texttt{last}}$ =
```
function last(l : list[@a]) : @a ⇐
  if(?empty(t),
     *,
     if(?empty(tl(l)),
        hd(l),
        last(tl(l))))
```

$D_{\texttt{but\_last}}$ =
```
function but_last(l : list[@a]) : list[@a] ⇐
  if(?empty(t),
     *,
     if(?empty(tl(l)),
        empty),
        add(hd(l), last(tl(l))))
```

$D_{\texttt{bubble}}$ =
```
function bubble(l : list[nat]) : list[nat] ⇐
  if(?empty(t),
     *,
     if(?empty(tl(l)),
        l),
        if(hd(l)>hd(tl(l)),
           add(hd(l), bubble(tl(l))),
           add(hd(tl(l)), bubble(add(hd(l), tl(tl(l))))))))
```

$D_{\texttt{bsort}}$ =
```
function bsort(l : list[@a]) : list[@a] ⇐
  if(?empty(t),
     empty,
     let b:=bubble(l) in
        add(last(b), bsort(but_last(b)))
     end)
```

Abbildung 12.3: Prozedurdefinitionen von `last`, `but_last`, `bubble` und `bsort`.

Beispiel 12.12. Sei P das Programm $\langle D_{\texttt{list}}, D_{\texttt{last}}, D_{\texttt{but\_last}}, D_{\texttt{bubble}}, D_{\texttt{bsort}}\rangle$, wobei $D_{\texttt{list}}$ die entsprechende Typoperatordefinition aus Abbildung 3.1 und $D_{\texttt{last}}$, $D_{\texttt{but\_last}}$, $D_{\texttt{bubble}}$ sowie $D_{\texttt{bsort}}$ die entsprechenden Prozedurdefinitionen der Abbildung 12.3 bezeichnen. Für die Gleichung

$$\texttt{last(bubble(l'))=hd(bsort(l'))}$$

des Lemmas

$$\begin{array}{l}\texttt{lemma bubble\_bsort} \Leftarrow \texttt{all l : list[nat]}\\ \quad\texttt{if(?empty(l), true, last(bubble(l'))=hd(bsort(l))}\end{array}$$

gilt dann

$$\texttt{hd(bsort(l'))} \gtrsim_{\mathrm{emb},P} \texttt{last(bubble(l'))}$$

und somit $\texttt{last(bubble(l'))} \not\gtrsim_P \texttt{hd(bsort(l'))}$. Da auch $\texttt{last(bubble(l'))} \not\gtrsim_P \texttt{hd(bsort(l'))}$ gilt, wird die Gleichung nicht gerichtet. Ohne die Bedingung (4) der Definition von $\gtrsim_P$ würde

$$\texttt{last(bubble(l'))} \succ_P \texttt{hd(bsort(l'))}$$

gelten und somit eine Auswertung des Terms

$$\texttt{if(?empty(k), 0, last(bubble(k)))}$$

mit Hilfe des Lemmas `bubble_bsort` und der Regel „*Assumption replacement*“ zu folgender Endlos-Auswertung führen:

$$\begin{array}{lll}
\mathcal{U} \vdash & \texttt{if(?empty(k), 0, }\underline{\texttt{last(bubble(k))}}\texttt{)} & \rightarrow\\
& \texttt{if(?empty(k), 0, if(}\underline{\texttt{?empty(k)}}\texttt{, last(bubble(k)), hd(bsort(k)))))} & \rightarrow\\
& \texttt{if(?empty(k), 0, }\underline{\texttt{if(false, last(bubble(k)), hd(bsort(k)))))}}\texttt{)} & \rightarrow\\
& \texttt{if(?empty(k), 0, hd(}\underline{\texttt{bsort(k)}}\texttt{))} & \rightarrow\\
& \texttt{if(?empty(k), 0, hd(if(}\underline{\texttt{?empty(k)}}\texttt{, empty, add(last(bubble(k)), ...))))} & \rightarrow\\
& \texttt{if(?empty(k), 0, hd(}\underline{\texttt{if(false, empty, add(last(bubble(k)), ...))))}}\texttt{)} & \rightarrow\\
& \texttt{if(?empty(k), 0, }\underline{\texttt{hd(add(last(bubble(k)), ...)))}}\texttt{)} & \rightarrow\\
& \texttt{if(?empty(k), 0, }\underline{\texttt{last(bubble(k))}}\texttt{)} & \rightarrow\\
& \quad\ldots & \vdots
\end{array}$$

Die Bedingung (4) verhindert also Endlos-Auswertungen aufgrund des Zusammenspiels der Regeln „*Assumption replacement*“ und „*Execute procedure call*“. □

Kapitel 13

Die „*Functionality*“-Regeln

Wie wir bereits in Abschnitt 6.2 kurz beschrieben haben, ist für jedes Funktionssymbol $f : \tau_1 \times \ldots \times \tau_n \to \tau \in \Sigma(P)$ mit nicht-boolschem Ergebnistyp die Quantifikation

$$\begin{array}{l}\texttt{all}\ \nu^*, x_1, y_1 : \tau_1, \ldots, x_n, y_n : \tau_n \\ \quad \texttt{if}(\mathit{AND}(x_1{=}y_1, \ldots, x_n{=}y_n), f(x_1, \ldots, x_n){=}f(y_1, \ldots, y_n), \texttt{true})\end{array} \tag{13.1}$$

gültig. Außerdem ist für Funktionssymbole $f : \tau_1 \times \ldots \times \tau_n \to \texttt{bool} \in \Sigma(P)$ mit boolschem Ergebnistyp die folgende Quantifikation gültig:

$$\begin{array}{l}\texttt{all}\ \nu^*, x_1, y_1 : \tau_1, \ldots, x_n, y_n : \tau_n \\ \quad \texttt{if}(\mathit{AND}(x_1{=}y_1, \ldots, x_n{=}y_n), \texttt{if}(f(x_1, \ldots, x_n), \\ \qquad\qquad\qquad\qquad f(y_1, \ldots, y_n), \\ \qquad\qquad\qquad\qquad \neg f(y_1, \ldots, y_n)), \texttt{true})\end{array} \tag{13.2}$$

Diese Quantifikationen drücken die Verträglichkeit der Gleichheit mit den anderen Funktionssymbolen des Programms aus (siehe Abschnitt 7.8). Um die Quantifikationen mit Hilfe der „*Assumption*“-Regeln während der symbolischen Auswertung anzuwenden, wäre es notwendig, für jedes Funktionssymbol eine entsprechende Quantifikationen zu definieren und die Gültigkeit der Quantifikation zu beweisen. Da dies wenig praktikabel ist, definieren wir in diesem Kapitel spezielle Auswertungsregeln zur Anwendung der Quantifikationen (13.1) und (13.2). Konkret definieren wir die folgenden Regeln:

$$\begin{array}{ll}\vdots & \vdots \\ 55. & \textit{Affirmative left non-boolean functionality} \\ 56. & \textit{Affirmative right non-boolean functionality} \\ 57. & \textit{Negative left non-boolean functionality} \\ 58. & \textit{Negative right non-boolean functionality} \\ 59. & \textit{Affirmative boolean functionality hypothesis} \\ 60. & \textit{Negative boolean functionality hypothesis} \\ \vdots & \vdots\end{array}$$

Bei diesen Regeln handelt es sich um Spezialisierungen der „*Assumption*“-Regeln für die Quantifikationen (13.1) und (13.2). Sie sind auf Basis der folgenden Vorgehensweisen definiert:

Vorgehensweise (nicht-boolsche Funktionssymbole):
Hinreichend für den Nachweis der Gültigkeit einer Gleichung

$$f(l_1,\ldots,l_n)\text{=}f(r_1,\ldots,r_n)$$

ist der Nachweis der Gültigkeit der Literale $l_i\text{=}r_i$ mit $i \in \{1,\ldots,n\}$. Hierzu werden die Literale symbolisch ausgewertet. Können alle Literale $l_i\text{=}r_i$ zu `true` ausgewertet werden, so darf die Gleichung korrekterweise durch `true` ersetzt werden.

Vorgehensweise (boolsche Funktionssymbole):
Hinreichend für den Nachweis der Gültigkeit bzw. der Ungültigkeit eines Atoms

$$f(l_1,\ldots,l_n)$$

ist der Nachweis der Gültigkeit der Literale $l_i\text{=}r_i$ mit $i \in \{1,\ldots,n\}$ für eine Hypothese $f(r_1,\ldots,r_n) \in H_{\mathrm{L}} \cup H_{\mathrm{G}}$ bzw. eine Hypothese $\neg f(r_1,\ldots,r_n) \in H_{\mathrm{L}} \cup H_{\mathrm{G}}$. Hierzu werden die Literale symbolisch ausgewertet. Können alle Literale $l_i\text{=}r_i$ zu `true` ausgewertet werden, so darf das Atom $f(l_1,\ldots,l_n)$ korrekterweise durch `true` bzw. `false` ersetzt werden.

Zur Definition der Regeln gehen wir wie folgt vor. In Abschnitt 13.1 definieren wir die Regeln zur Anwendung der Quantifikation (13.1) und in Abschnitt 13.2 definieren wir die Regeln zur Anwendung der Quantifikation (13.2).

13.1 Nicht-boolsche Funktionssymbole

Für die Quantifikation (13.1) definieren wir zunächst die folgende vorläufige Auswertungsregel:

$$\frac{f(l_1,\ldots,l_n)\text{=}^{\boxed{A}} f(r_1,\ldots,r_n)}{\mathbf{env}_{\mathcal{U}'}(AND(d_1,\ldots,d_n))}, \qquad \begin{array}{|c|}\hline \text{S-Lim.} \\ \hline A\text{:}\quad s \\ \hline \end{array}$$

falls $s > 0$, $f \in \Sigma(P) \setminus \Sigma^{\mathrm{cons}}(P)$, $p \neq \ominus$ und die folgenden Bedingungen erfüllt sind:

(1) $D = \{l_1\text{=}r_1,\ldots,l_n\text{=}r_n\}$.

(2) $s_{\mathrm{new}}(1,|D|+1) \geq 1$.

(3) $\{d_1,\ldots,d_n\} = \mathbf{pol}_{\oplus}(\mathbf{a}_{A'}(D))$.

(4) $\forall k \in \{1,\ldots,n\}$ gilt $\mathcal{U}' \vdash d_k \rightarrow \mathtt{true}$.

(13.3)

Hierbei sind die A-Umgebung $\mathcal{U}'$ und die A-Annotation A' wie folgt gegeben:

$$\begin{array}{|c|}\hline \text{Reg.} \\ \hline \mathcal{U}'\text{:}\quad no\text{-}execute(\mathcal{R}) \\ \hline \end{array} \qquad \begin{array}{|cc|}\hline A_{\mathrm{top}} & \text{S-Lim.} \\ \hline A'\text{:} & s_{\mathrm{new}}(1,|D|+1) \\ \hline \end{array}$$

Es ist zu beachten, dass im Gegensatz zu den „*Assumption*“-Regeln in Regel (13.3) zur Vereinfachung der Subgoals aus Effizienzgründen keine Prozeduraufrufe ausgewertet werden. Die Berechnung des neuen Search-Limits in (13.3) stimmt hingegen mit der Berechnung des Search-Limits in den „*Assumption*“-Regeln überein,

da $|D|+1$ der Kardinalität der Klausel

$$\{\neg(x'_1{=}y'_1), \ldots, \neg(x'_n{=}y'_n), f(x'_1, \ldots, x'_n){=}f(y'_1, \ldots, y'_n)\}.$$

entspricht.

Wir wollen nun in den folgenden Abschnitten die vorläufige Regeldefinition (13.3) um einige Aspekte erweitern. In Abschnitt 13.1.1 zeigen wir, wie wir mit Hilfe der lokalen und globalen Hypothesen die Quantifikation (13.1) auch dann auf eine Gleichung $l{=}r$ anwenden können, wenn nicht beide Seiten der Gleichung mit dem gleichen Funktionssymbol beginnen. Weiter zeigen wir, wie die Quantifikation (13.1) verwendet werden kann, um die Ungültigkeit einer Gleichung zu zeigen. In Abschnitt 13.1.2 beschreiben wir dann, wie geschachtelte Anwendungen der Quantifikation (13.1) in der Regel (13.3) berücksichtigt werden können. Anschließend zeigen wir in Abschnitt 13.1.3, wie durch die Verwendung der Kommutativität und der Assoziativität des Funktionssymbols f zusätzliche Anwendungen der Regel (13.3) ermöglicht werden können. Die durch die Abschnitte 13.1.1-13.1.3 definierten Erweiterungen führen zu verschiedenen Versionen der Regel (13.3), die wir dann in Abschnitt 13.1.4 abschließend definieren.

13.1.1 Verwendung der Hypothesen

Betrachten wir den folgenden Term:

$$\texttt{if(x=y, if(plus(x, y)=z, z=plus(x, x), true), true)}\,.$$

Dieser Term kann mit Hilfe der Regel (13.3) nicht ausgewertet werden. Wir können aber die linke Seite der Gleichung $\texttt{z=plus(x, x)}$ mit Hilfe der Transitivität der Gleichheit und der lokalen Hypothese $\texttt{plus(x, y)=z}$ durch $\texttt{plus(x, y)}$ ersetzen und so eine Anwendung der Regel (13.3) ermöglichen:

$$\begin{array}{lll}
\mathcal{U} \vdash & \texttt{if(x=y, if(plus(x, y)=z, }\underline{\texttt{plus(x, y)=plus(x, x)}}\texttt{, true), true)} & \rightarrow \\
& \texttt{if(x=y, if(plus(x, y)=z, if(}\underline{\texttt{x=x}}\texttt{, y=x, false), true), true)} & \rightarrow \\
& \texttt{if(x=y, if(plus(x, y)=z, }\underline{\texttt{if(true, y=x, false)}}\texttt{, true), true)} & \rightarrow \\
& \texttt{if(x=y, if(plus(x, y)=z, }\underline{\texttt{y=x}}\texttt{, true), true)} & \rightarrow \\
& \texttt{if(x=y, }\underline{\texttt{if(plus(x, y)=z, true, true)}}\texttt{, true)} & \rightarrow \\
& \underline{\texttt{if(x=y, true, true)}} & \rightarrow \\
& \texttt{true}. &
\end{array}$$

Um solche symbolische Auswertungen zu erlauben, erweitern wir Regel (13.3) so, dass wir für einen Term der Form $f(l_1, \ldots, l_n){=}t$ bzw. $t{=}f(r_1, \ldots, r_n)$ die globalen und lokalen Hypothesen nach einem Term der Form $t{=}f(r_1, \ldots, r_n)$ bzw. $f(l_1, \ldots, l_n){=}t$ durchsuchen und dann die Subgoals d_i auf Basis der Gleichung $f(l_1, \ldots, l_n){=}f(r_1, \ldots, r_n)$ erzeugen. Wir formulieren daher die folgende Modifikation:

Modifikation (Verwendung positiver Hypothesen):
Die Regel (13.3) wird allgemein für Gleichungen der Form

$$t{=}f(l_1, \ldots, l_n) \quad \text{bzw.} \quad f(l_1, \ldots, l_n){=}t$$

angewendet, sofern eine Hypothese der Form $f(r_1, \ldots, r_n){=}t$ bzw. der Form $t{=}f(r_1, \ldots, r_n)$ in der lokalen oder globalen Hypothesenmenge enthalten ist.

Existiert für einen Term $f(l_1,\ldots,l_n)$=t eine globale oder lokale Hypothese der Form $\neg(t$=$f(r_1,\ldots,r_n))$ und können wir die Gleichungen l_i=r_i zu `true` vereinfachen, so können wir offensichtlich den Term $f(l_1,\ldots,l_n)$=t durch `false` ersetzen. Wir definieren daher zusätzlich die folgende Modifikation der Regel (13.3):

Modifikation (Verwendung negativer Hypothesen):
Eine Gleichung der Form

$$t\text{=}f(s_1,\ldots,s_n) \quad \text{bzw.} \quad f(s_1,\ldots,s_n)\text{=}t$$

wird durch die Regel (13.3) durch `false` ersetzt, sofern eine Hypothese der Form $\neg(f(r_1,\ldots,r_n)$=$t)$ bzw. der Form $\neg(t$=$f(r_1,\ldots,r_n))$ in der lokalen oder globalen Hypothesenmenge enthalten ist und die restlichen Anwendungsbedingungen der Regel erfüllt sind.

13.1.2 Verwendung der kompletten Termstruktur

Um mit Hilfe der Regel (13.3) die Gültigkeit des Terms

$$\texttt{if(x=y,plus(x,plus(x,x))=plus(x,plus(x,y)),true)} \tag{13.4}$$

nachzuweisen, ist es notwendig Regel (13.3) zweimal anzuwenden:[1]

$$\begin{aligned}
\mathcal{U} \vdash\ & \texttt{if(x=y,}\underline{\texttt{plus(x,plus(x,x))=}^{256}\texttt{plus(x,plus(x,y))}}\texttt{,true)} && \rightarrow\\
& \texttt{if(x=y,if(}\underline{\texttt{x=}^{64}\texttt{x}}\texttt{,plus(x,x)=}^{64}\texttt{plus(x,y),false),true)} && \rightarrow\\
& \texttt{if(x=y,}\underline{\texttt{if(true,plus(x,x)=}^{64}\texttt{plus(x,y),false)}}\texttt{,true)} && \rightarrow\\
& \texttt{if(x=y,}\underline{\texttt{plus(x,x)=}^{64}\texttt{plus(x,y)}}\texttt{,true)} && \rightarrow\\
& \texttt{if(x=y,if(}\underline{\texttt{x=}^{16}\texttt{x}}\texttt{,x=}^{16}\texttt{y,false),true)} && \rightarrow\\
& \texttt{if(x=y,}\underline{\texttt{if(true,x=}^{16}\texttt{y,false)}}\texttt{,true)} && \rightarrow\\
& \texttt{if(x=y,}\underline{\texttt{x=}^{16}\texttt{y}}\texttt{,true)} && \rightarrow\\
& \underline{\texttt{if(x=y,true,true)}} && \rightarrow\\
& \texttt{true}
\end{aligned}$$

Die Auswertung von Termen wie (13.4) ist also durch die maximale Suchtiefe $depth_{\max}$ beschränkt. Bei einer maximalen Suchtiefe von $depth_{\max} = 4$ können wir daher einen Term wie

$$\left.\begin{array}{l}\texttt{if(x=y,}\\ \quad\texttt{plus(a,plus(b,plus(c,plus(d,plus(e,x)))))=}\\ \quad\quad\texttt{plus(a,plus(b,plus(c,plus(d,plus(e,y))))),}\\ \quad\texttt{true)}\end{array}\right\} = E \tag{13.5}$$

mit Hilfe von Regel (13.3) nicht auswerten. Um die Beschränkung durch die maximale Suchtiefe zu umgehen, modifizieren wir daher Regel (13.3) dahingehend, dass durch einmalige Anwendung der Regel für die Gleichung E in (13.5) nicht mehr die Gleichungen

$$\texttt{a=a},$$
$$\texttt{plus(b,plus(c,plus(d,plus(e,x))))=plus(b,plus(c,plus(d,plus(e,y))))}$$

[1] Wir haben der Übersichtlichkeit halber lediglich die Search-Limits, der für die Auswertung interessanten Gleichungen dargestellt.

als Subgoals erzeugt werden, sondern die Gleichungen, die sich aufgrund der gemeinsamen „`plus`-Termstruktur“ der linken und rechten Seite der Gleichung E ergeben:

$$\texttt{a=a}, \quad \texttt{b=b}, \quad \texttt{c=c}, \quad \texttt{d=d}, \quad \texttt{e=e}, \quad \texttt{x=y}.$$

Um diese Gleichungen zu bilden, definieren wir die folgende Funktion:

$$\begin{array}{rcll} \mathit{flat}_f(t,x) & := & \langle t{=}x\rangle, & \\ \mathit{flat}_f(x,t) & := & \langle x{=}t\rangle, & \\ \mathit{flat}_f(h(l_1,\ldots,l_n),g(r_1,\ldots,r_m)) & := & \langle h(l_1,\ldots,l_n){=}g(r_1,\ldots,r_m)\rangle & \text{falls } g \neq h, \\ \mathit{flat}_f(f(l_1,\ldots,l_n),f(r_1,\ldots,r_n)) & := & \mathit{flat}_f(l_1,r_1) \oplus \ldots \oplus \mathit{flat}_f(l_n,r_n).^2 & \end{array}$$

Wir definieren daher die folgende Modifikation der Regel (13.3):

Modifikation (Verwendung der kompletten Termstruktur):
Um die komplette Termstruktur bei der Anwendung der Regel (13.3) zu berücksichtigen, werden die Subgoals der Regel durch den folgenden Ausdruck berechnet:

$$\mathit{flat}_f(f(l_1,\ldots,l_n),f(r_1,\ldots,r_n)).$$

13.1.3 Verwendung der Kommutativität und Assoziativität

Für eine Prozedur f, die als assoziativ und kommutativ bekannt ist, müssen wir uns nicht auf die durch $\mathit{flat}_f(f(l_1,\ldots,l_n),f(r_1,\ldots,r_n))$ erzeugten Gleichungen

$$\langle t_1{=}s_1,\ldots,t_m{=}s_m\rangle \tag{13.6}$$

als Subgoals beschränken. Vielmehr können wir die Subgoals der Regel (13.3) aufgrund beliebiger Permutation der Argumente

$$t_1,\ldots,t_m \text{ und } s_1,\ldots,s_m$$

bilden. Da jedoch die Überprüfung aller Permutationen zu aufwendig ist, betrachten wir neben den Gleichungen (13.6) lediglich die Gleichungen als Subgoals, deren linken und rechten Seiten einen minimalen syntaktisch Unterschied aufweisen. Wir formulieren daher die folgende Modifikation:

Modifikation (Kommutativität und Assoziativität):
Für kommutative und assoziative Funktionssymbole f betrachten wir zur Anwendung der Regel (13.3) neben den Gleichungen

$$\mathit{flat}_f(f(l_1,\ldots,l_n),f(r_1,\ldots,r_n))$$

auch die Gleichungen der Termstruktur mit den geringsten syntaktischen Unterschieden.

Zur Berechnung der Gleichungen verwenden wir die in Anhang C definierte Funktion $\mathit{perm\text{-}flat}_{\mathcal{U},f}$. Mit dieser Funktion können wir dann nachfolgend die vollständigen Regeldefinitionen angeben. Um diese Regeldefinitionen kompakt darstellen zu können, führen wir für die zu überprüfenden Subgoals die folgende Funktion ein:

$$\begin{array}{lcll} \mathit{eq\text{-}list}_{\mathcal{U},f}(t,r) & = & \{\mathit{flat}_f(t,r)\} & \text{falls } f \notin \Sigma_C \vee f \notin \Sigma_A, \\ \mathit{eq\text{-}list}_{\mathcal{U},f}(t,r) & = & \{\mathit{flat}_f(t,r), \mathit{perm\text{-}flat}_f(t,r)\} & \text{falls } f \in \Sigma_C \wedge f \in \Sigma_A. \end{array}$$

[2] $\oplus$ bezeichnet die Konkatenation von Listen (siehe Anhang A.1).

13.1.4 Vollständige Regeldefinitionen

Um die in den Abschnitte 13.1.1-13.1.3 definierten Modifikationen in die Regel (13.3) einzufügen ist es zweckmäßig verschiedene Versionen der Regel zu definieren. Wir führen daher in diesem Abschnitt die folgenden Regeln ein:

$\vdots$ $\quad$ $\vdots$
53. Affirmative left non-boolean functionality
54. Affirmative right non-boolean functionality
55. Negative left non-boolean functionality
56. Negative right non-boolean functionality
$\vdots$ $\quad$ $\vdots$

Die beiden ersten Regeln ersetzen eine Gleichung $l{=}r$ aufgrund der Quantifikation (13.1) durch `true` und die beiden anderen Regeln ersetzen die Gleichung aufgrund der Quantifikation durch `false`. Die Regeln sind wie folgt definiert:

55. Affirmative left non-boolean functionality

$$\frac{f(l_1,\ldots,l_n) =^{A} t}{\mathsf{env}_{\mathcal{U}'}(AND(d_1,\ldots,d_m))}, \qquad \begin{array}{|c|c|}\hline & \text{S-Lim.} \\ \hline A\text{:} & s \\ \hline \end{array}$$

falls $s > 0$, $f \in \Sigma(P) \setminus \Sigma^{\text{cons}}(P)$, $p \neq \ominus$ und die folgenden Bedingungen erfüllt sind:

(1) $t = f(r_1,\ldots,r_n)$ oder $\exists h \in_{\simeq_{\mathcal{U}}} H_{\text{L}} \cup H_{\text{G}}$ mit $h = t{=}f(r_1,\ldots,r_n)$.

(2) $D \in \mathit{eq\text{-}list}_{\mathcal{U},f}(f(l_1,\ldots,l_n), f(r_1,\ldots,r_n))$.

(3) $s_{\text{new}}(1, |D|+1) \geq 1$.

(4) $\langle d_1,\ldots,d_m\rangle = \mathsf{pol}_{\oplus}(\mathsf{a}_{A'}(D))$.

(5) $\forall k \in \{1,\ldots,m\}$ gilt $\mathcal{U}' \vdash d_k \rightarrow \texttt{true}$.

Hierbei sind die A-Umgebung $\mathcal{U}'$ und die A-Annotation A' wie folgt gegeben:

Reg.
$\mathcal{U}'$: $\mathit{no\text{-}execute}(\mathcal{R})$

A_{top}	S-Lim.
A':	$s_{\text{new}}(1, \lvert D\rvert + 1)$

56. Affirmative right non-boolean functionality

$$\frac{t =^{A} f(l_1,\ldots,l_n)}{\mathsf{env}_{\mathcal{U}'}(AND(d_1,\ldots,d_m))}, \qquad \begin{array}{|c|c|}\hline & \text{S-Lim.} \\ \hline A\text{:} & s \\ \hline \end{array}$$

falls $s > 0$, $f \in \Sigma(P) \setminus \Sigma^{\text{cons}}(P)$, $p \neq \ominus$ und die folgenden Bedingungen erfüllt sind:

(1) $\exists h \in_{\simeq_{\mathcal{U}}} H_{\text{L}} \cup H_{\text{G}}$ mit $h = t{=}f(r_1,\ldots,r_n)$.

... der Rest der Regel ist entsprechend der Regel 55 definiert ...

57. Negative left non-boolean functionality

$$\frac{f(l_1,\ldots,l_n) \overset{A}{=} t}{\mathbf{env}_{\mathcal{U}'}(\neg(AND(d_1,\ldots,d_m)))},$$

	S-Lim.
A:	s

falls $s > 0$, $f \in \Sigma(P) \setminus \Sigma^{\text{cons}}(P)$, $p \neq \oplus$ und die folgenden Bedingungen erfüllt sind:

(1) $\exists \neg h \in_{\simeq_{\mathcal{U}}} H_{\text{L}} \cup H_{\text{G}}$ mit $h = t{=}f(r_1,\ldots,r_n)$.

(2) $D \in \mathit{eq\text{-}list}_{\mathcal{U},f}(f(l_1,\ldots,l_n), f(r_1,\ldots,r_n))$.

(3) $s_{\text{new}}(1, |D|+1) \geq 1$.

(4) $\langle d_1,\ldots,d_m\rangle = \mathbf{pol}_{\oplus}(\mathbf{a}_{A'}(D))$.

(5) $\forall k \in \{1,\ldots,m\}$ gilt $\mathcal{U}' \vdash d_k \rightarrow \texttt{true}$.

Hierbei sind die A-Umgebung $\mathcal{U}'$ und die A-Annotation A' wie folgt gegeben:

Reg.
$\mathcal{U}'$: $\mathit{no\text{-}execute}(\mathcal{R})$

A_{top}	S-Lim.
A':	$s_{\text{new}}(1, \lvert D\rvert + 1)$

58. Negative right non-boolean functionality

$$\frac{t \overset{A}{=} f(l_1,\ldots,l_n)}{\mathbf{env}_{\mathcal{U}'}(\neg(AND(d_1,\ldots,d_m)))},$$

	S-Lim.
A:	s

falls $s > 0$, $f \in \Sigma(P) \setminus \Sigma^{\text{cons}}(P)$, $p \neq \oplus$ und die folgenden Bedingungen erfüllt sind:

(1) $\exists \neg h \in_{\simeq_{\mathcal{U}}} H_{\text{L}} \cup H_{\text{G}}$ mit $h = t{=}f(r_1,\ldots,r_n)$.

... der Rest der Regel ist entsprechend der Regel 57 definiert ...

13.2 Boolsche Funktionssymbole

Für Funktionssymbole mit boolschem Ergebnistyp definieren wir aufgrund der Quantifikation (13.2) die folgenden Regeln:

59. Affirmative boolean functionality hypothesis

$$\frac{f^{\,A}(l_1,\ldots,l_n)}{\mathbf{env}_{\mathcal{U}'}(AND(d_1,\ldots,d_n))},$$

	S-Lim.
A:	s

falls $s > 0$, $n > 0$, $f^{\,A}(l_1,\ldots,l_n) \in \mathcal{ATerm}_{\mathtt{bool}}(P)$, $f \in \Sigma(P) \setminus (\Sigma^{\mathrm{case}}(P) \cup \Sigma_0^{=})$, $p \neq \ominus$ und die folgenden Bedingungen erfüllt sind:

(1) $\exists h \in H_{\mathrm{L}} \cup H_{\mathrm{G}}$ mit $h = f(r_1,\ldots,r_n)$.

(2) $D = \langle l_1{=}r_1,\ldots,l_n{=}r_n\rangle$.

(3) $s_{\mathrm{new}}(1,|D|+1) \geq 1$.

(4) $\langle d_1,\ldots,d_n\rangle = \mathbf{pol}_{\oplus}(\mathbf{a}_{A'}(D))$.

(5) $\forall k \in \{1,\ldots,n\}$ gilt $\mathcal{U}' \vdash d_k \rightarrow \mathtt{true}$.

Hierbei sind die A-Umgebung $\mathcal{U}'$ und die A-Annotation A' wie folgt gegeben:

	Reg.
$\mathcal{U}'$:	$no\text{-}execute(\mathcal{R})$

A_{top}	S-Lim.
A':	$s_{\mathrm{new}}(1,\lvert D\rvert+1)$

60. Negative boolean functionality hypothesis

$$\frac{f^{\,A}(l_1,\ldots,l_n)}{\mathbf{env}_{\mathcal{U}'}(\neg(AND(d_1,\ldots,d_n)))},$$

	S-Lim.
A:	s

falls $s > 0$, $n > 0$, $f^{\,A}(l_1,\ldots,l_n) \in \mathcal{ATerm}_{\mathtt{bool}}(P)$, $f \in \Sigma(P) \setminus (\Sigma^{\mathrm{case}}(P) \cup \Sigma_0^{=})$, $p \neq \ominus$ und die folgenden Bedingungen erfüllt sind:

(1) $\exists \neg h \in H_{\mathrm{L}} \cup H_{\mathrm{G}}$ mit $h = f(r_1,\ldots,r_n)$.

... der Rest der Regel ist entsprechend der Regel 59 definiert ...

13.3 Anmerkungen zur Implementierung

In ✓`eriFun` existieren weitere „*Functionality*“-Regeln. Diese Regeln haben wir in Anhang E definiert. In älteren Versionen des Auswertungskalküls haben sich diese Regeln zwar als nützlich erwiesen, bei einer erneuten Untersuchung der Regeln hat sich jedoch gezeigt, dass sie in unseren Fallstudien nicht mehr verwendet werden. Wir haben sie daher nicht in unsere Definition des Auswertungskalküls aufgenommen.

Teil III

Erweiterung und Verwendung

Kapitel 14

Der erweiterte Auswertungskalkül

Der Auswertungskalkül aus Abschnitt 7.1 terminiert nicht. Das bedeutet, wir können mit Hilfe der definierten Regeln Endlos-Auswertungen konstruieren.[1] Da wir jedoch, wie in Abschnitt 5.1 beschrieben, die Terminierung des Auswertungskalküls gewährleisten müssen, ist es notwendig, die Definition des Auswertungskalküls einer Überarbeitung zu unterziehen. Hierzu erweitern wir in Abschnitt 14.1 die A-Umgebungen um den so genannten Schrittzähler und definieren eine überarbeitete Version des Auswertungskalküls . Der Schrittzähler begrenzt die maximale Anzahl der Auswertungsschritte einer Auswertung und gewährleistet so die Terminierung der Auswertung. Es ist hierbei zu beachten, dass die Ergebnisse der Auswertungen, die aufgrund des Schrittzählers terminieren im Allgemeinen für die weitere Beweisführung im HPL-Kalkül wenig geeignet sind. Im Gegensatz dazu sind die Ergebnisse der Auswertungen, die aufgrund des *Search-Limits*, des *Unfold-Limits*, der Labels, etc. terminieren für die weitere Beweisführung gut geeignet. Dies ist auch nicht weiter verwunderlich, da dies ja gerade der Zweck dieser Steuerinformationen ist. Der Schrittzähler gewährleistet somit lediglich, dass die Implementierung des Auswertungskalküls terminiert und der Benutzer des ✓eriFun-Systems so die Gelegenheit bekommt die berechnete Auswertung zu verwerfen, um andere, sinnvollere Beweisregeln auf die ursprüngliche Sequenz anzuwenden.

Auf Basis des überarbeiteten Auswertungskalküls aus Abschnitt 14.1 definieren wir in Abschnitt 14.2 den erweiterten Auswertungskalkül. Dieser erweiterte Auswertungskalkül setzt nach einer erfolgreichen symbolischen Auswertung die Annotationen des Ergebnisterms zurück und wertet den Term erneut aus. Durch dieses Vorgehen werden zusätzliche Auswertungen ermöglicht. Schließlich beschreiben wir in Abschnitt 14.3, wie der erweiterte Auswertungskalkül in den *Computed Proof Rules* des HPL-Kalküls verwendet wird.

14.1 Terminierung des Auswertungskalküls

Um die Terminierung des Auswertungskalküls zu gewährleisten, erweitern wir A-Umgebungen um die Anzahl der maximal zulässigen Auswertungsschritte einer symbolischen Auswertung. Für jeden Auswertungsschritt dekrementieren wir diesen Zähler und wenden eine Auswertungsregel nur dann an, wenn der Zähler größer als 0 ist. Um dieses Verfahren zu realisieren ist es notwendig, die Definitionen 6.1 und 7.1 aus den Abschnitten 6.1 und 7.1 zu überarbeiten.

[1] Ein Beispiel für eine Endlos-Auswertung ist in Beispiel F.1 in Anhang F angegeben.

Definition 14.1 (Überarbeitete Definition der A-Umgebungen)**.** Sei P ein terminierendes Programm. Sei weiter

$$\langle P, \Sigma, \Sigma_{\mathrm{C}}, \Sigma_{\mathrm{A}}, \mathcal{Q}, H_{\mathrm{L}}, H_{\mathrm{G}}, H_{\mathrm{I}}, C, \mathcal{R}, \delta, \rangle$$

eine A-Umgebung im Sinne von Definition 6.1 und *steps* eine ganze Zahl. Als *Auswertungsumgebung* (oder kurz *A-Umgebung*) von P wird von nun an ein Tupel der Form

$$\langle P, \Sigma, \Sigma_{\mathrm{C}}, \Sigma_{\mathrm{A}}, \mathcal{Q}, H_{\mathrm{L}}, H_{\mathrm{G}}, H_{\mathrm{I}}, C, \mathcal{R}, \delta, \mathit{steps} \rangle.$$

bezeichnet. Die Zahl *steps* heißt dann *Schrittzähler der A-Umgebung.* □

Für eine A-Umgebung

$$\mathcal{U} = \langle P, \Sigma, \Sigma_{\mathrm{C}}, \Sigma_{\mathrm{A}}, \mathcal{Q}, H_{\mathrm{L}}, H_{\mathrm{G}}, H_{\mathrm{I}}, C, \mathcal{R}, \delta, \mathit{steps} \rangle$$

schreiben wir $\mathcal{U} > 0$, wenn der Schrittzähler *steps* größer als 0 ist. Weiter bezeichnen wir mit $\mathcal{U} - n$ die folgende A-Umgebung:

$$\langle P, \Sigma, \Sigma_{\mathrm{C}}, \Sigma_{\mathrm{A}}, \mathcal{Q}, H_{\mathrm{L}}, H_{\mathrm{G}}, H_{\mathrm{I}}, C, \mathcal{R}, \delta, \mathit{steps} - n \rangle.$$

Definition 14.2 (Überarbeitete Definition des Auswertungskalküls)**.** Sei P ein terminierendes Programm und $\mathcal{U}$ eine A-Umgebung für P. Der *Auswertungskalkül* (oder kurz *A-Kalkül*) ist dann wie folgt definiert:

(1) **Sprache:** *Entsprechend Definition 7.1*

(2) **Inferenzregeln:** *Entsprechend Definition 7.1*

(3) **Deduktion:** Wir schreiben $\mathcal{U} \vdash t \to t'$ falls $\mathcal{U} > 0$ gilt und eine Regel existiert, die eine Ableitung der Form

$$\frac{t}{t'}$$

bezüglich $\mathcal{U}$ ermöglicht, wobei die Regeln in der durch die Regelnummern festgelegten Reihenfolge überprüft werden. Für eine Folge $\langle t_i \rangle_{i \in \{1,\ldots,n\}}$ von annotierten Termen mit $(\mathcal{U} - (i-1)) \vdash t_i \to t_{i+1}$ für alle $i = 1, \ldots, n-1$ schreiben wir $\mathcal{U} \vdash t_1 \to \ldots \to t_n$ bzw. $\mathcal{U} \vdash t_1 \to^* t_n$.

□

Der so definierte Auswertungskalkül terminiert offensichtlich. In ✓eriFun wird die maximale Länge der Auswertungen durch die Konstante $\mathit{steps}_{\max}$ festgelegt. Für die Version 3.0 des Systems ist diese Konstante auf den Wert 300000 gesetzt. Da alle in unseren Fallstudien aufgetretenen symbolischen Auswertungen deutlich kürzer als 300000 waren, stellt dieser Wert aus praktischer Sicht keine Beschränkung für die Länge der symbolischen Auswertungen dar.

14.2 Zurücksetzen der Annotationen

Einige der Auswertungsregeln setzen die Unfold- und Search-Limits von Teiltermen aus Effizienzgründen auf 0. Ein Beispiel für eine solche Regel ist die Regel „*Unfold left equality argument*“. Weiter setzt der Auswertungskalkül das Instanz-Flag aller neu erzeugten `case`-Terme auf $\bot$. Das bedeutet, dass die entsprechenden Bedingungen nicht zur Instantiierung der Subgoals in den Regeln „*Affirmative assumption*“ und „*Negative assumption*“ verwendet werden. Für eine erfolgreiche Auswertung eines Goal-Terms ist es daher unter Umständen sinnvoll, die Unfold- und Search-Limits sowie die Instanz-Flags des Goal-Terms nach einer symbolischen Auswertung auf die Werte $u_{\max}$, $s_{\max}$ bzw. $\top$ zurückzusetzen und den Goal-Term erneut auszuwerten. Betrachten wir dazu ein Beispiel:

```
if(?empty(tl(k)),
   if(?empty(tl(l)),
      true,
         if(hd(l)>hd(tl(l)),
         true,
         if(ordered(tl(l)),
            if(hd(k)>hd(l),
               if(?empty(merge^0(k, tl(l))),
                  true,
                  if(hd(l)>hd(merge^0(k, tl(l))), false, true)),
               true),
            true))),
   if(hd(k)>hd(tl(k)),
      true,
      if(ordered(tl(k)),
         if(?empty(tl(l)),
            if(hd(k)>hd(l),
               true,
               if(?empty(merge^0(tl(k), l)),
                  true,
                  if(hd(k)>hd(merge^0(tl(k), l)), false, true))),
            if(hd(l)>hd(tl(l)),
               true,
               if(ordered(tl(l)),
                  if(hd(k)>hd(l),
                     if(?empty(merge^0(k, tl(l))),
                        true,
                        if(hd(l)>hd(merge^0(k, tl(l))), false, true)),
                     if(?empty(merge^0(tl(k), l)),
                        true,
                        if(hd(k)>hd(merge^0(tl(k), l)), false, true))),
                  true))),
         true)))
```

Abbildung 14.1: Ergebnis der symbolischen Auswertung des Goal-Terms der Sequenz (14.2) aus Beispiel 14.3. Es werden der Übersichtlichkeit halber ausschließlich die Unfold-Limits der Prozeduraufrufe von `merge` dargestellt.

Beispiel 14.3. Sei P das Programm $\langle D_{\texttt{list}}, D_{\texttt{ordered}}, D_{\texttt{dis\_ev}}, D_{\texttt{dis\_odd}}, D_{\texttt{merge}}\rangle$, wobei $D_{\texttt{list}}$ die entsprechende Typoperatordefinition aus Abbildung 3.1 bezeichnet und $D_{\texttt{ordered}}$, $D_{\texttt{dis\_ev}}$, $D_{\texttt{dis\_odd}}$ und $D_{\texttt{merge}}$ die entsprechenden Prozedurdefinitionen aus den Abbildungen 9.3 und 11.1 bezeichnen. Sei weiter die Menge der generalisierten Relationenbeschreibungen gegeben durch

$$\mathit{grds}(\texttt{dis\_ev}) = \mathit{grds}(\texttt{dis\_odd}) = \mathit{grds}(\texttt{ordered}) = \{R_1\}$$

und

$$\mathit{grds}(\texttt{merge}) = \{R_2\},$$

wobei gilt

$$\begin{aligned} R_1 &= \{\langle\langle \neg\texttt{?empty(k)}\rangle, \{\{\texttt{k/tl(k)}\}\}\rangle\}, \\ R_2 &= \{\langle\langle \neg\texttt{?empty(k)}, \neg\texttt{?empty(l)}\rangle, \{\{\texttt{l/tl(l)}, \texttt{k/k}\}, \{\texttt{l/l}, \texttt{k/tl(k)}\}\}\rangle\}. \end{aligned}$$

Betrachten wir nun den Beweis des Lemmas

$$\begin{aligned} &\texttt{lemma merge\_keeps\_ordered} \Leftarrow \texttt{all k : list[nat], l : list[nat]} \\ &\quad \texttt{if(ordered(k), if(ordered(l), ordered(merge(k, l)), true), true)}, \end{aligned} \tag{14.1}$$

wobei wir die Gültigkeit des Lemmas

$$\begin{aligned} &\texttt{lemma >\_asymmetric} \Leftarrow \texttt{all x : nat, y : nat} \\ &\quad \texttt{if(x>y, if(y>x, false, true), true)} \end{aligned}$$

als gegeben voraussetzen. Für das Lemma (14.1) schlägt der Teilterm $\texttt{merge(k, l)}$ eine Induktion über die Relationenbeschreibung R_2 vor. Für diese Induktion ergibt sich der Schrittfall

$$\begin{aligned} &\{\neg\texttt{?empty(k)}, \neg\texttt{?empty(l)}\}, \{\mathit{ih}_1, \mathit{ih}_2\} \vdash \\ &\texttt{if(ordered(k), if(ordered(l), ordered(merge(k, l)), true), true)} \end{aligned} \tag{14.2}$$

mit den folgenden Induktionshypothesen:

$$\begin{aligned} \mathit{ih}_1 &= \texttt{if(ordered(tl(k)), if(ordered(l), ordered(merge(tl(k), l)), true), true)}, \\ \mathit{ih}_2 &= \texttt{if(ordered(k), if(ordered(tl(l)), ordered(merge(k, tl(l))), true), true)}. \end{aligned}$$

Eine symbolische Auswertung des Goal-Terms unter der globalen Hypothesenmenge

$$\{\neg\texttt{?empty(k)}, \neg\texttt{?empty(l)}\}$$

und der Multimenge

$$\mathcal{Q} = \{\!\{1 * \mathit{ih}_1, 1 * \mathit{ih}_2, 2 * \texttt{all x : nat, y : nat if(x>y, if(y>x, false, true), true)}\}\!\}$$

liefert den in Abbildung 14.1 dargestellten Term. Alle Prozeduraufrufe von `merge` können durch zurücksetzen des Unfold-Limits auf $u_{\max}$ mit Hilfe der Unfold-Regel „*Unfold structure test argument*“ und „*Unfold selector argument*“ ausgewertet werden. Durch Auswertung dieser Prozeduraufrufe können wir dann den Term zu `true` vereinfachen. □

Es ist zu beachten, dass wir ausschließlich die Annotationen von Goal-Termen zurücksetzen möchten und nicht von Termen, die aufgrund von Anwendungsbedingungen ausgewertet werden. So soll beispielsweise das Unfold-Limit während der Auswertung der Subgoals in „*Affirmative assumption*“ und „*Negative assumption*“ nicht zurückgesetzt werden. Dies wäre wenig sinnvoll, da wir das Unfold-Limit für die Subgoals ganz bewusst aus Effizienzgründen auf 0 gesetzt haben. Um für Goal-Terme das Zurücksetzen der Annotationen zu ermöglichen, definieren wir daher auf

Basis des Auswertungskalküls den *erweiterten Auswertungskalkül*. Dieser erweiterte Auswertungskalkül wertet einen Term mit Hilfe des Auswertungskalküls solange aus, bis der Goal-Term vollständig ausgewertet ist. Danach werden die Annotationen zurückgesetzt und der Term wird erneut mit Hilfe des Auswertungskalküls ausgewertet. Der erweiterte Auswertungskalkül ist wie folgt definiert:

Definition 14.4 (Erweiterter Auswertungskalkül)**.** Sei P ein terminierendes Programm und $\mathcal{U}$ eine A-Umgebung für P. Der *erweiterte Auswertungskalkül* (oder kurz *erweiterte A-Kalkül*) ist dann wie folgt definiert:

(1) **Sprache:** Die A-Terme über P.

(2) **Inferenzregeln:**

$$\frac{t}{t'}, \quad \text{falls } \mathcal{U} \vdash t \to t' \qquad\qquad \frac{t}{\mathbf{a}_{A_{\max}}(t)}, \quad \text{falls } t\downarrow_{\mathcal{U}} = t \text{ und } t \neq \mathbf{a}_{A_{\max}}(t)$$

(3) **Deduktion:** Wir schreiben $\mathcal{U} \vdash t \Rightarrow t'$ falls $\mathcal{U} > 0$ gilt und eine Regel existiert, die eine Ableitung der Form

$$\frac{t}{t'}$$

bezüglich $\mathcal{U}$ ermöglicht. Für eine Folge $\langle t_i \rangle_{i \in \{1,\ldots,n\}}$ von annotierten Termen mit $(\mathcal{U} - (i-1)) \vdash t_i \Rightarrow t_{i+1}$ für alle $i = 1, \ldots, n-1$ schreiben wir $\mathcal{U} \vdash t_1 \Rightarrow \ldots \Rightarrow t_n$ bzw. $\mathcal{U} \vdash t_1 \Rightarrow^* t_n$. Gilt $\mathcal{U} \vdash t \Rightarrow^* t'$ und existiert kein t'' mit $\mathcal{U} \vdash t' \Rightarrow t''$, so schreiben wir $\mathcal{U} \vdash t \Rightarrow^! t'$ bzw. $t' = t\Downarrow_{\mathcal{U}}$.

□

Wir verwenden dann im HPL-Kalkül zur Auswertung von Goal-Termen den erweiterten Auswertungskalkül.

14.3 Verwendung des Auswertungskalküls

Der erweiterte Auswertungskalkül wird vom HPL-Kalkül durch die „*Computed Proof Rules*“ verwendet, um den Goal-Term der aktuellen Sequenz zu vereinfachen (siehe Abschnitt 2.2). Wir wollen in diesem Abschnitt kurz darstellen, wie die A-Umgebungen für die einzelnen „*Computed Proof Rules*“ definiert sind. Hierbei gehen wir davon aus, dass wir den Goal-Term der Sequenz

$$H, IH \vdash g \tag{14.3}$$

auswerten möchten. Weiter gehen wir von den folgenden Annahmen aus:

- Σ bezeichnet die Termsignatur aller Prozeduren des aktuellen annotierten Programms, deren Terminierung durch das ✓`eriFun`-System bewiesen wurde.
- Σ' bezeichnet die Termsignatur aller nicht-rekursiv definierten Prozeduren des aktuellen annotieren Programms, deren Terminierung durch das ✓`eriFun`-System bewiesen wurde.
- Σ_{C} bzw. Σ_{A} enthalten alle Prozeduren, deren Kommutativität bzw. Assoziativität durch die Verifikation eines entsprechenden Lemmas im aktuellen annotierten Programm bewiesen wurde.
- $\mathcal{Q}$ enthält die Induktionshypothesen der Sequenz (14.3) sowie die Quantifikationen der verifizierten Lemmata, die durch den Lemma-Filter (siehe Kapitel 15) ausgewählt wurden. Die Anzahl der einzelnen Quantifikationen in der Multimenge entspricht den Festlegungen aus Kapitel 11 auf Seite 175.

Aufbauend auf diesen Annahmen sind dann die A-Umgebungen der „*Computed Proof Rules*“ wie folgt definiert:

- **Simplification:** Diese Beweisregel verwendet zur symbolischen Auswertung die Typoperator- und Prozedurdefinitionen, die durch den Lemma-Filter ausgewählten Lemmata sowie die Hypothesen und Induktionshypothesen der Sequenz. Es ergibt sich dadurch die folgende A-Umgebung:

$$\langle P, \Sigma, \Sigma_{\mathrm{C}}, \Sigma_{\mathrm{A}}, \mathcal{Q}, \emptyset, H, \emptyset, \{\top\}, \mathcal{R}_0, \emptyset, \mathit{steps}\rangle.$$

- **Weak Simplification:** Die durch diese Regel definierte symbolische Auswertung unterscheidet sich von der symbolischen Auswertung mit „*Simplification*“ lediglich dahingehend, dass nur nicht-rekursiv definierte Prozeduren ausgewertet werden. Es ergibt sich dann die folgende A-Umgebung:

$$\langle P, \Sigma', \Sigma_{\mathrm{C}}, \Sigma_{\mathrm{A}}, \mathcal{Q}, \emptyset, H, \emptyset, \{\top\}, \mathcal{R}_0, \emptyset, \mathit{steps}\rangle.$$

- **Normalization:** Die „*Normalization*“-Regel schränkt die symbolische Auswertung dahingehend ein, dass gar keine Prozeduren ausgewertet werden. Das bedeutet, dass zur symbolischen Auswertung lediglich die Typoperatordefinitionen, die durch den Lemma-Filter ausgewählten Lemmata sowie die Hypothesen und Induktionshypothesen der Sequenz verwendet werden. Es ergibt sich dadurch die folgende A-Umgebung:

$$\langle P, \emptyset, \Sigma_{\mathrm{C}}, \Sigma_{\mathrm{A}}, \mathcal{Q}, \emptyset, H, \emptyset, \{\top\}, \mathcal{R}_0, \emptyset, \mathit{steps}\rangle.$$

- **Weak Normalization:** Die durch diese Regel definierte symbolische Auswertung wertet weder Prozeduren aus, noch werden Lemmata und Induktionshypothesen zur Auswertung der Terme verwendet. Das bedeutet, dass die Regel lediglich einfache Umformungen durchführt. Es ergibt sich dann die folgende A-Umgebung:

$$\langle P, \emptyset, \Sigma_{\mathrm{C}}, \Sigma_{\mathrm{A}}, \emptyset, \emptyset, H, \emptyset, \{\top\}, \mathcal{R}_0, \emptyset, \mathit{steps}\rangle.$$

Kapitel 15

Der Lemma-Filter

In den Abschnitten 14.3 haben wir den so genannten Lemma-Filter erwähnt. Beim Lemma-Filter handelt es sich um eine Heuristik, die für die „*Computed Proof Rules*“ des HPL-Kalküls die Menge der zu verwendenden Lemmata bestimmt. Je mehr Lemmata während einer symbolischen Auswertung zur Verfügung stehen, desto größer ist der Suchraum und desto ineffizienter ist die symbolische Auswertung. Werden jedoch während der symbolischen Auswertung zu wenig Lemmata betrachtet, so kann der Goal-Term der Sequenz unter Umständen nur unzureichend ausgewertet werden. In diesen Fällen sind dann Benutzerinteraktionen zum Beweis der Gültigkeit der Sequenz notwendig. Die genaue Festlegung der während der symbolischen Auswertung zu verwendenden Lemmata ist somit für das effektive Arbeiten mit dem ✓eriFun-System von entscheidender Bedeutung.

Während der symbolischen Auswertung des Goal-Terms einer Sequenz

$$H, IH \vdash g \tag{15.1}$$

wenden wir Lemmata mit Hilfe der Regeln „*Affirmative assumption*“, „*Negative assumption*“ und „*Assumption replacement*“ an. Wir verwenden dabei jedoch nicht die Lemmata direkt, sondern die Klauseln, die mit den Lemmata assoziiert sind. Hierzu ermitteln wir für ein Literal einer Klausel und den aktuell auszuwertenden Term einen Matcher und werten dann die entsprechend erzeugten Subgoals symbolisch aus (siehe Abschnitt 11). Um vor einer symbolischen Auswertung eines Goal-Terms zu ermitteln, welche Lemmata des aktuellen annotierten Programms für die symbolische Auswertung geeignet sind, überprüft der Lemma-Filter auf Basis der Prozeduren der Sequenz des Goal-Terms und der Prozeduren der Klauseln der Lemmata, ob eine Verwendung der Klauseln durch die Regeln „*Affirmative assumption*“, „*Negative assumption*“ und „*Assumption replacement*“ möglich erscheint. Betrachten wir dazu ein Beispiel:

Beispiel 15.1. Sei P das Programm $\langle D_{\texttt{plus}}, D_{\texttt{minus}} \rangle$, wobei $D_{\texttt{plus}}$ und $D_{\texttt{minus}}$ die entsprechenden Prozedurdefinitionen aus den Abbildungen 8.2 und 9.1 bezeichnen. Betrachten wir nun die symbolische Auswertung des Goal-Terms der folgenden Sequenz:

$$\{\neg\texttt{?0(x)}\}, \{\texttt{all y : nat if(plus(pred(x), y)=pred(x), ?0(y), true)}\} \vdash \texttt{if(plus(x, y)=x, ?0(y), true)}. \tag{15.2}$$

Wir gehen davon aus, dass die Gültigkeit des Lemmas

$$\texttt{lemma minus\_plus} \Leftarrow \texttt{all x, y, z : nat if(plus(x, y)=z, minus(z, y)=x, true)}$$

$$
\begin{array}{lll}
\mathcal{U} \vdash & \mathtt{if}(\underline{\mathtt{plus}(\mathtt{x},\mathtt{y})}{=}\mathtt{x}, \mathtt{?0}(\mathtt{y}), \mathtt{true}) & \rightarrow \\
& \vdots & \vdots \\
& \mathtt{if}(\underline{\mathtt{succ}(\mathtt{plus}(\mathtt{pred}(\mathtt{x}),\mathtt{y}))}{=}\mathtt{x}, \mathtt{?0}(\mathtt{y}), \mathtt{true}) & \rightarrow \\
& \mathtt{if}(\underline{\mathtt{succ}(\mathtt{plus}(\mathtt{pred}(\mathtt{x}),\mathtt{y}))}{=}\underline{\mathtt{succ}(\mathtt{pred}(\mathtt{x}))}, \mathtt{?0}(\mathtt{y}), \mathtt{true}) & \rightarrow \\
& \mathtt{if}(\mathtt{plus}(\mathtt{pred}(\mathtt{x}),\mathtt{y}){=}\mathtt{pred}(\mathtt{x}), \mathtt{?0}(\mathtt{y}), \mathtt{true}) & \rightarrow \\
& \mathtt{if}(\mathtt{plus}(\mathtt{pred}(\mathtt{x}),\mathtt{y}){=}\mathtt{pred}(\mathtt{x}), \underline{\mathtt{?0}(\mathtt{y})}, \mathtt{true}) & \rightarrow \\
& \vdots & \vdots \\
& \mathtt{if}(\mathtt{plus}(\mathtt{pred}(\mathtt{x}),\mathtt{y}){=}\mathtt{pred}(\mathtt{x}), \mathtt{true}, \mathtt{true}) & \rightarrow \\
& \mathtt{true} &
\end{array}
$$

Abbildung 15.1: Symbolische Auswertung des Goal-Terms der Sequenz (15.2) aus Beispiel 15.1 unter Verwendung der Hypothese und der Induktionshypothese.

---

bewiesen ist und somit das Lemma bzw. die mit dem Lemma assoziierte Klauselmenge

$$\{\underbrace{\{\neg(\mathtt{plus}(\mathtt{x}',\mathtt{y}')=\mathtt{z}'), \mathtt{minus}(\mathtt{z}',\mathtt{y}')=\mathtt{x}'\}}_{C}\}$$

während der symbolischen Auswertung verwendet werden darf. Betrachten wir die Klausel C und den Goal-Term der Sequenz (15.2) genauer, so stellen wir fest, dass aufgrund der Tatsache, dass im Goal-Term kein `minus`-Term enthalten ist, für das Literal

$$\mathtt{minus}(\mathtt{z}',\mathtt{y}')=\mathtt{x}'$$

und einen beliebigen Teilterm des Goal-Terms mit Sicherheit kein Matcher berechnet werden kann. Das bedeutet, eine Verwendung des Lemmas `minus_plus` während der Auswertung des Goal-Terms ist nur aufgrund des Literals

$$\neg(\mathtt{plus}(\mathtt{x}',\mathtt{y}')=\mathtt{z}')$$

möglich. Eine solche Verwendung würde jedoch einen entsprechenden `minus`-Term als Subgoal erzeugen. Da jedoch in der Sequenz (15.2) kein `minus`-Term enthalten ist, ist eine erfolgreiche Auswertung des entsprechenden Subgoals ohne Verwendung weiterer Lemmata ausgesprochen unwahrscheinlich. Der Lemma-Filter entfernt daher für die symbolische Auswertung des Goal-Terms von (15.2) das Lemma `minus_plus` aus der Menge der zu verwendenden Lemmata. Wie die Abbildung 15.1 zeigt, ist für die symbolische Auswertung des Goal-Terms das Lemma `minus_plus` tatsächlich nicht erforderlich und der Lemma-Filter hat somit korrekterweise das Lemma aus der Menge der zu verwendenden Lemmata entfernt. □

Das Beispiel lässt vermuten, dass die Prozeduren der Sequenz ein geeignetes Kriterium darstellen, um zu entscheiden, ob die Klauseln bzw. die entsprechenden Lemmata während der symbolischen Auswertung des Goal-Terms sinnvoll verwendet werden können. Wir definieren daher auf Basis der Prozeduren der aktuellen Sequenz zwei Klassen von Klauseln und verwenden für die symbolische Auswertung des Goal-Terms einer Sequenz ein Lemma nur dann, falls eine der Klauseln des Lemmas zu einer der beiden Klassen gehört.

Das Kapitel gliedert sich wie folgt: In Abschnitt 15.1 definieren wir zunächst die beiden Klassen von Klauseln. Anschließend definieren wir dann auf Basis der

```
D_msort = function msort(k : list[nat]) : list[nat] ⇐
            if(?empty(k),
               empty
               if(?empty(tl(k)),
                  k,
                  merge(msort(dis_ev(k)),msort(dis_odd(k)))))
```

Abbildung 15.2: Prozedurdefinition für `msort`.

beiden Klassen den Lemma-Filter. In Abschnitt 15.2 beschreiben wir, wie mit Hilfe des Lemma-Filters die Multimenge $\mathcal{Q}$ der A-Umgebungen für die „*Computed Proof Rules*“ berechnet wird. Abschließend illustrieren wir in Abschnitt 15.3 die Auswirkungen des Lemma-Filters auf die symbolischen Auswertungen der „*Computed Proof Rules*“.

15.1 Definition des Lemma-Filters

Sind alle Prozeduren einer Klausel in der aktuellen Sequenz enthalten, so spricht nichts gegen die Verwendung der Klausel bzw. des Lemmas während der symbolischen Auswertung des Goal-Terms. Wir bezeichnen solche Klauseln als *schwache Klauseln*. Schwache Klauseln werden grundsätzlich während der symbolischen Auswertung des Goal-Terms betrachtet. Sind in einer Klausel gerichtete Gleichungen enthalten, d.h. Literale der Form $l{=}r$ mit $l \succ_P r$ und sind in allen Literalen $lit \in C \setminus \{l{=}r\}$ sowie in l nur Prozeduren enthalten, die in der Sequenz enthalten sind, so spricht aufgrund der Regel „*Assumption replacement*“ auch nichts gegen die Verwendung dieser Klausel. Wir bezeichnen auch solche Klauseln als schwache Klauseln. Betrachten wir ein Beispiel:

Beispiel 15.2. Sei P das Programm

$$\langle D_{\texttt{list}}, D_{\texttt{plus}}, D_{\texttt{occurs}}, D_{\texttt{merge}}, D_{\texttt{dis\_ev}}, D_{\texttt{dis\_odd}}, D_{\texttt{msort}}\rangle,$$

wobei $D_{\texttt{list}}$ die Typoperatordefinition aus Abbildung 3.1 und $D_{\texttt{plus}}$, $D_{\texttt{occurs}}$, $D_{\texttt{merge}}$, $D_{\texttt{dis\_ev}}$, $D_{\texttt{dis\_odd}}$ sowie $D_{\texttt{msort}}$ die entsprechenden Prozedurdefinitionen aus den Abbildungen 8.2, 12.1, 11.1 und 15.2 bezeichnen. Betrachten wir die symbolische Auswertung des Goal-Terms der folgenden Sequenz:

$$\{\neg\texttt{?empty(k)}, \neg\texttt{?empty(tl(k))}\}, \{ih_1, ih_2\} \vdash g \tag{15.3}$$

mit

```
ih1 = all n : nat occurs(n, dis_ev(k))=occurs(n, msort(dis_ev(k))),
ih2 = all n : nat occurs(n, dis_odd(k))=occurs(n, msort(dis_odd(k)))
```

und

```
g = if(n=hd(k),
       if(n=hd(tl(k)),
          succ(succ(occurs(n, tl(tl(k)))))
           =occurs(n, merge(msort(dis_ev(k)), msort(dis_odd(k)))),
          succ(occurs(n, tl(tl(k))))
           =occurs(n, merge(msort(dis_ev(k)), msort(dis_odd(k))))),
       if(n=hd(tl(k)),
          succ(occurs(n, tl(tl(k))))
           =occurs(n, merge(msort(dis_ev(k)), msort(dis_odd(k)))),
          occurs(n, tl(tl(k)))
           =occurs(n, merge(msort(dis_ev(k)), msort(dis_odd(k)))))).
```

Wir gehen davon aus, dass die Gültigkeit der folgenden Lemmata durch das System bereits bewiesen ist:

$$\begin{array}{l}\texttt{lemma plus\_commutative} \Leftarrow \texttt{all y : nat, x : nat}\\ \quad \texttt{plus(x, y)=plus(y, x)},\\[2ex] \texttt{lemma occurs\_merge} \Leftarrow \texttt{all k : list[nat], l : list[nat], n : nat}\\ \quad \texttt{occurs(n, merge(k, l))=plus(occurs(n, k), occurs(n, l))},\\[2ex] \texttt{lemma occurs\_distribute} \Leftarrow \texttt{all k : list[nat], n : nat}\\ \quad \texttt{occurs(n, k)=plus(occurs(n, dis\_ev(k)), occurs(n, dis\_odd(k)))}.\end{array} \tag{15.4}$$

Wir können daher die mit den Lemmata assoziierten Klauseln

$$\{\underbrace{\{\texttt{plus(x', y')=plus(y', x')}\}}_{C_1}\},$$

$$\{\underbrace{\{\texttt{occurs(n', merge(k', l'))=plus(occurs(n', k'), occurs(n', l'))}\}}_{C_2}\},$$

$$\{\underbrace{\{\texttt{occurs(n', k')=plus(occurs(n', dis\_ev(k')), occurs(n, dis\_odd(k')))}\}}_{C_3}\}$$

zur symbolischen Auswertung des Goal-Terms g verwenden, wobei zu beachten ist, dass die Gleichung in C_2 durch $\succ_P$ von links nach rechts gerichtet wird und die Gleichung in C_3 von rechts nach links. Es ist weiter zu beachten, dass jede Klausel die Prozedur `plus` enthält und diese Prozedur nicht in der Sequenz (15.3) enthalten ist. Für die Literale der Klauseln und die Teilterme des Terms g kann daher mit Sicherheit kein Matcher berechnet werden. Die Klauseln sind daher für eine Verwendung durch die Regeln „*Affirmative assumption*“ und „*Negative assumption*“ während der symbolischen Auswertung von g ungeeignet. Da jedoch

$$\texttt{occurs(n', merge(k', l'))} \quad \succ_P \quad \texttt{plus(occurs(n', k'), occurs(n', l'))}$$

gilt, spricht aufgrund der in der Sequenz enthaltenen Prozeduren nichts gegen einen Matcher der linken Seite des Literals von C_2 und eines Teilterms von g und somit nichts gegen eine Verwendung der Klausel C_2 durch die Regel „*Assumption replacement*“. Da durch eine solche Verwendung der Klausel die Prozedur `plus` in den Goal-Term eingeführt wird, spricht dann auch nichts mehr gegen eine Verwendung der restlichen Lemmata während der symbolischen Auswertung. Wie die in Abbildung 15.3 skizzierte symbolische Auswertung zeigt, sind alle Lemmata für die erfolgreiche Auswertung von g erforderlich. □

Zur formalen Definition schwacher Klauseln benötigen wir die folgende Funktion:

$$proc_P^{=}(lit) \quad = \quad \begin{cases} proc_P(l) & \text{falls } lit \simeq (l{=}r) \text{ und } l \succ_P r,\\ proc_P(lit) & \text{sonst.}\end{cases}$$

Schwache Klauseln definieren wir dann wie folgt.

Definition 15.3 (Schwache Klauseln)**.** Sei P ein terminierendes Programm, C eine Klausel über P und F ein Menge von Prozeduren von P. Die Klausel C heißt *schwache Klausel bzgl.* F gdw. für alle Literale $lit \in C$ gilt $proc_P^{=}(lit) \subseteq F$ □

Bezeichnet F die Menge der Prozeduren der Sequenz (15.3) aus Beispiel 15.2, so ist allein Klausel C_2 eine schwache Klausel bzgl. F.

$\mathcal{U} \vdash$

$\ldots$ succ(succ(occurs(n, tl(tl(k)))))=occurs(n, merge(msort(dis_ev(k)), msort(dis_odd(k)))) $\ldots$ $\xrightarrow{(1)}$ (mit C_2)

$\ldots$ succ(succ(occurs(n, tl(tl(k)))))=if(true, plus(occurs(n, msort(dis_ev(k))), occurs(n, msort(dis_odd(k)))), $\ldots$) $\ldots$ $\xrightarrow{(2)}$

$\ldots$ succ(succ(occurs(n, tl(tl(k)))))=plus(occurs(n, msort(dis_ev(k))), occurs(n, msort(dis_odd(k)))) $\ldots$ $\xrightarrow{(3)}$ (mit IH)

$\ldots$ succ(succ(occurs(n, tl(tl(k)))))=plus(if(true, occurs(n, dis_ev(k)), $\ldots$), occurs(n, msort(dis_odd(k)))) $\ldots$ $\xrightarrow{(4)}$

$\vdots$ $\vdots$

$\ldots$ succ(succ(occurs(n, tl(tl(k)))))=plus(occurs(n, dis_ev(k)), occurs(n, msort(dis_odd(k)))) $\ldots$ $\xrightarrow{(5)}$

$\vdots$ $\vdots$

$\ldots$ succ(succ(occurs(n, tl(tl(k)))))=plus(succ(occurs(n, dis_ev(tl(k)))), occurs(n, msort(dis_odd(k)))) $\ldots$ $\xrightarrow{(6)}$ (mit IH)

$\vdots$ $\vdots$

$\ldots$ succ(succ(occurs(n, tl(tl(k)))))=plus(succ(occurs(n, dis_ev(tl(k)))), succ(occurs(n, dis_odd(tl(tl(k)))))) $\ldots$ $\xrightarrow{(7)}$

$\vdots$ $\vdots$

$\ldots$ succ(occurs(n, tl(tl(k))))=plus(occurs(n, dis_ev(tl(k))), succ(occurs(n, dis_odd(tl(tl(k)))))) $\ldots$ $\xrightarrow{(8)}$

$\vdots$ $\vdots$

$\ldots$ occurs(n, tl(tl(k)))=plus(occurs(n, dis_ev(tl(k))), occurs(n, dis_odd(tl(tl(k))))) $\ldots$ $\xrightarrow{(9)}$ (mit C_3)

$\ldots$ occurs(n, tl(tl(k)))=if(true, occurs(n, tl(tl(k))), $\ldots$) $\ldots$ $\xrightarrow{(10)}$

$\ldots$ occurs(n, tl(tl(k)))=occurs(n, tl(tl(k))) $\ldots$ $\xrightarrow{(11)}$

$\ldots$ true $\ldots$

Abbildung 15.3: Symbolische Auswertung des Goal-Terms der Sequenz (15.3) aus Beispiel 15.2. Die in den Klammern angegebenen Klauseln bzw. Quantifikationen geben an, welche Klausel bzw. Quantifikation im jeweiligen Auswertungsschritt verwendet wurde.

$$
\begin{aligned}
D_{\texttt{app}} \;:=\;& \texttt{function app(k : list[@a], l : list[@a]) : list[@a]} \;\Leftarrow\\
& \quad \texttt{if(?empty(k),}\\
& \qquad \texttt{l,}\\
& \qquad \texttt{add(hd(k), app(tl(k), l)))}
\end{aligned}
$$

$$
\begin{aligned}
D_{\texttt{smaller}} \;:=\;& \texttt{function smaller(n : nat, k : list[nat]) : list[nat]} \;\Leftarrow\\
& \quad \texttt{if(?empty(k),}\\
& \qquad \texttt{empty,}\\
& \qquad \texttt{if(hd(k)>n,}\\
& \qquad\quad \texttt{smaller(n, tl(k)),}\\
& \qquad\quad \texttt{add(hd(k), smaller(n, tl(k)))))}
\end{aligned}
$$

$$
\begin{aligned}
D_{\texttt{larger}} \;:=\;& \texttt{function larger(n : nat, k : list[nat]) : list[nat]} \;\Leftarrow\\
& \quad \texttt{if(?empty(k),}\\
& \qquad \texttt{empty,}\\
& \qquad \texttt{if(hd(k)>n,}\\
& \qquad\quad \texttt{add(hd(k), larger(n, tl(k))),}\\
& \qquad\quad \texttt{larger(n, tl(k))))}
\end{aligned}
$$

$$
\begin{aligned}
D_{\texttt{qsort}} \;:=\;& \texttt{function qsort(k : list[nat]) : list[nat]} \;\Leftarrow\\
& \quad \texttt{if(?empty(k),}\\
& \qquad \texttt{empty,}\\
& \qquad \texttt{app(qsort(smaller(hd(k), tl(k))),}\\
& \qquad\quad \texttt{add(hd(k), qsort(larger(hd(k), tl(k)))))))}
\end{aligned}
$$

$$
\begin{aligned}
D_{\texttt{upper\_bound}} \;:=\;& \texttt{function upper\_bound(k : list[nat], n : nat) : bool} \;\Leftarrow\\
& \quad \texttt{if(?empty(k),}\\
& \qquad \texttt{true,}\\
& \qquad \texttt{if(hd(k)>n,}\\
& \qquad\quad \texttt{false,}\\
& \qquad\quad \texttt{upper\_bound(tl(k), n)))}
\end{aligned}
$$

$$
\begin{aligned}
D_{\texttt{lower\_bound}} \;:=\;& \texttt{function lower\_bound(n : nat, k : list[nat]) : bool} \;\Leftarrow\\
& \quad \texttt{if(?empty(k),}\\
& \qquad \texttt{true,}\\
& \qquad \texttt{if(n>hd(k),}\\
& \qquad\quad \texttt{false,}\\
& \qquad\quad \texttt{lower\_bound(n, tl(k))))}
\end{aligned}
$$

Abbildung 15.4: Prozedurdefinitionen für `app`, `smaller`, `larger`, `qsort`, `upper_bound` und `lower_bound`.

$\mathcal{U} \vdash$ `ordered(app(qsort(`S`), add(hd(k), qsort(`L`))))` $\xrightarrow{(1)}$

`if(ordered(app(...,...))`
`  true,`
`  if(ordered(qsort(`S`)),`
`    if(upper_bound(qsort(`S`), hd(k)),`
`      if(lower_bound(hd(k), qsort(`L`)), ordered(qsort(`L`)), false),`
`      false),`
`    false))`

$\xrightarrow{(2)}$ (mit C_5)

⋮ ⋮

`if(ordered(app(...,...))`
`  true,`
`  if(true,`
`    if(upper_bound(qsort(`S`), hd(k)),`
`      if(lower_bound(hd(k), qsort(`L`)), ordered(qsort(`L`)), false),`
`      false),`
`    false))`

$\xrightarrow{(3)}$

`if(ordered(app(...,...))`
`  true,`
`  if(upper_bound(qsort(`S`), hd(k)),`
`    if(lower_bound(hd(k), qsort(`L`)), ordered(qsort(`L`)), false),`
`    false))`

$\xrightarrow{(4)}$ (mit C_2)

`if(ordered(app(...,...))`
`  true,`
`  if(if(upper_bound(qsort(`S`), hd(k)), true, upper_bound(`S`, hd(k))),`
`    if(lower_bound(hd(k), qsort(`L`)), ordered(qsort(`L`)), false),`
`    false))`

$\xrightarrow{(5)}$ (mit C_4)

`if(ordered(app(...,...))`
`  true,`
`  if(if(upper_bound(qsort(`S`), hd(k)),`
`      true,`
`      if(upper_bound(`S`, hd(k)), true, true)),`
`    if(lower_bound(hd(k), qsort(`L`)), ordered(qsort(`L`)), false),`
`    false))`

$\xrightarrow{(6)}$

⋮ ⋮

`if(ordered(app(...,...))`
`  true,`
`  if(lower_bound(hd(k), qsort(`L`)), ordered(qsort(`L`)), false))`

$\xrightarrow{(7)}$ (mit C_1)

⋮ ⋮

`if(ordered(app(...,...)), true, true)` $\xrightarrow{(8)}$

`true`

Abbildung 15.5: Symbolische Auswertung des Goal-Terms der Sequenz (15.5) aus Beispiel 15.4. Hierbei haben wir den Term `smaller(hd(k), tl(k))` durch S und den Term `larger(hd(k), tl(k))` durch L abgekürzt. Weiter haben wir die Klauseln bzw. Induktionshypothesen angegeben, die im jeweiligen Auswertungsschritt verwendet werden.

Die ausschließliche Verwendung von Lemmata mit schwachen Klauseln während der symbolischen Auswertung eines Goal-Terms würde bedeuten, dass alle Prozeduren, die durch die Verwendung einer Klausel durch die Regeln „*Affirmative assumption*“ und „*Negative Assumption*“ in den Goal-Term eingeführt werden, bereits im Goal-Term vorhanden sein müssen. Abstrakt formuliert bedeutet das, dass zum Beweis eines Goal-Terms nur „Begriffe“ verwendet werden, die bereits im Goal-Term vorhanden sind. Dass es für den Beweis eines Goal-Terms durchaus sinnvoll ist, neue „Begriffe“ in den Goal-Term einzuführen, zeigt das folgende Beispiel:

Beispiel 15.4. Sei P das Programm

$$\langle D_{\texttt{list}}, D_{\texttt{app}}, D_{\texttt{ordered}}, D_{\texttt{smaller}}, D_{\texttt{larger}}, D_{\texttt{qsort}}, D_{\texttt{upper\_bound}}, D_{\texttt{lower\_bound}} \rangle,$$

wobei $D_{\texttt{list}}$ die entsprechende Typoperatordefinition aus Abbildung 3.1 und $D_{\texttt{app}}$, $D_{\texttt{ordered}}$, $D_{\texttt{smaller}}$, $D_{\texttt{larger}}$, $D_{\texttt{qsort}}$, $D_{\texttt{upper\_bound}}$ sowie $D_{\texttt{lower\_bound}}$ die entsprechenden Prozedurdefinitionen aus den Abbildungen 9.3 und 15.4 bezeichnen. Betrachten wir nun die symbolische Auswertung des Goal-Terms der folgenden Sequenz:

$$\{\neg\texttt{?0(k)}\}, \{ih_1, ih_2\} \vdash g. \tag{15.5}$$

Hierbei gilt

$$\begin{array}{rcl} ih_1 & = & \texttt{ordered(qsort(larger(hd(k),tl(k))))} \\ ih_2 & = & \texttt{ordered(qsort(smaller(hd(k),tl(k))))} \end{array}$$

und

$$\begin{array}{rcl} g & = & \texttt{ordered(app(qsort(smaller(hd(k),tl(k))),} \\ & & \qquad \texttt{add(hd(k),qsort(larger(hd(k),tl(k)))))).} \end{array}$$

Wir gehen davon aus, dass die Gültigkeit der folgenden Lemmata durch das System bereits bewiesen ist:

```
lemma lower_bound_qsort ⇐ all n : nat, k : list[nat]
  if(lower_bound(n, k), lower_bound(n, qsort(k)), true),

lemma upper_bound_qsort ⇐ all n : nat, k : list[nat]
  if(upper_bound(n, k), upper_bound(n, qsort(k)), true),

lemma larger_lower_bound ⇐ all n : nat, k : list[nat]
  lower_bound(n, larger(n, k)),

lemma smaller_upper_bound ⇐ all k : list[nat], n : nat
  upper_bound(smaller(n, k), n),

lemma ordered_append ⇐ all k : list[nat], l : list[nat], n : nat
  if(ordered(k),
     if(upper_bound(k, n),
        if(lower_bound(n, l),
           if(ordered(l), ordered(app(k, add(n, l))), true),
           true),
        true),
     true).
```

Die mit diesen Lemmata assoziierten Klauselmengen lauten wie folgt:

$$\{\underbrace{\{\neg\texttt{lower\_bound(n', k')}, \texttt{lower\_bound(n', qsort(k'))}\}}_{C_1}\},$$

$$\underbrace{\{\{\neg\texttt{upper\_bound}(\texttt{n}',\texttt{k}'),\texttt{upper\_bound}(\texttt{n}',\texttt{qsort}(\texttt{k}'))\}\}}_{C_2},$$

$$\underbrace{\{\{\texttt{lower\_bound}(\texttt{n}',\texttt{larger}(\texttt{n}',\texttt{k}'))\}\}}_{C_3},$$

$$\underbrace{\{\{\texttt{upper\_bound}(\texttt{smaller}(\texttt{n}',\texttt{k}'),\texttt{n}')\}\}}_{C_4},$$

$$\underbrace{\{\{\neg\texttt{ordered}(\texttt{k}'),\ \neg\texttt{ordered}(\texttt{l}'),\ \neg\texttt{upper\_bound}(\texttt{k}',\texttt{n}'),\ \neg\texttt{lower\_bound}(\texttt{n}',\texttt{l}'),\ \texttt{ordered}(\texttt{app}(\texttt{k}',\texttt{add}(\texttt{n}',\texttt{l}')))\}\}}_{C_5}$$

Mit Hilfe dieser Klauselmengen kann die Gültigkeit von (15.5) nachgewiesen werden (siehe Abbildung 15.5). Hierbei ist zu beachten, dass alle Klauseln der Klauselmengen für die symbolische Auswertung erforderlich sind. Da keine der Klauseln schwach bzgl. der im Goal-Term g enthaltenen Prozeduren ist, folgt, dass die Beschränkung der symbolischen Auswertung auf schwache Klauseln den Suchraum zu stark einschränkt. □

Wir müssen also für eine erfolgreiche symbolische Auswertung eines Goal-Terms auch Klauseln betrachten, die zusätzliche Prozeduren enthalten. Wir führen hierzu die Klasse der *starken Klauseln* ein.

Definition 15.5 (Starke Klauseln)**.** Sei P ein terminierendes Programm, C eine Klausel über P und F ein Menge von Prozeduren. Die Klausel C heißt *starke Klausel bzgl.* F gdw. ein Literal $lit \in C$ existiert mit $|proc_P^=(lit)| > 1$ und $proc_P^=(lit) \subseteq F$. □

Die Klausel C_5 aus Beispiel 15.4 ist eine starke Klausel bzgl. der Prozeduren der Sequenz des Beispiels. Es ist zu beachten, dass Definition 15.5 Klauseln wie

$$\{\neg(\texttt{plus}(\texttt{x}',\texttt{y}')=\texttt{z}'),\texttt{minus}(\texttt{z}',\texttt{y}')=\texttt{x}'\}$$

aufgrund der Bedingung $|proc_P^=(l)| > 1$ explizit ausschließt. Dadurch verhindern wir, dass die Klausel in Beispiel 15.1 für die Auswertung des Goal-Terms von (15.2) verwendet wird.

Nachdem wir nun die beiden Klassen von Klauseln eingeführt haben, definieren wir den eigentlichen Lemma-Filter.

Definition 15.6 (Lemma-Filter)**.** Sei P ein Programm, L eine Menge von Lemmata über P und F eine Menge von Prozeduren. Die Funktion $filter_F$ ist wie folgt definiert:

$$filter_F(L) \quad := \quad \{lem \in L \mid \exists C \in \mathcal{C}_{lem} \text{ mit } C \text{ ist schwache oder starke Klausel bzgl. } F\}$$

Die n-ten Vervollständigung $filter_F(n, L)$ der Menge $filter_F(L)$ ist dann gegeben durch

(1) $filter_F(0, L) := filter_F(L)$,

(2) $filter_F(n+1, L) := filter_{F \cup proc_P(filter_F(n,L))}(L)$.

Der *Lemma-Filter* filter_F^* ist dann als Vervollständigung der Menge $\mathit{filter}_F(L)$ definiert:

$$\mathit{filter}_F^*(L) := \bigcup_{n \in \mathbb{N}} \mathit{filter}_F(n, L).$$

□

Bezeichnet F die Menge aller Prozeduren der aktuellen Sequenz, so stellt Bedingung (1) der Definition der n-ten Vervollständigung sicher, dass alle Lemmata lem während der symbolischen Auswertung betrachtet werden, deren assoziierte Klauselmenge $\mathcal{C}_{\mathit{lem}}$ ein schwaches oder starkes Lemma bzgl. F enthalten. Bedingung (2) gewährleistet, dass die Prozeduren, die durch schwache oder starke Klauseln eingeführt werden, vom Lemma-Filter berücksichtigt werden. Für Beispiel 15.2 gilt beispielsweise $\mathit{filter}_F(0, L) = \{C_2\}$ und $\mathit{filter}_F(1, L) = \{C_1, C_2, C_3\}$, wobei F die Menge aller Prozeduren der Sequenz (15.3) bezeichnet und L die Menge der im Beispiel definierten Lemmata. Man erkennt, dass erst in der Menge $\mathit{filter}_F(1, L)$ die für eine vollständige Auswertung relevanten Klauseln enthalten sind. In Beispiel 15.4 würde ohne Bedingung (2) der Lemma-Filter lediglich das Lemma `ordered_append` als relevant erkennen. Dieses Lemma reicht jedoch für die vollständige symbolische Auswertung des Goal-Terms von (15.5) nicht aus. Durch Bedingung (2) werden auch die anderen Lemmata des Beispiels 15.4 als relevant erkannt.

15.2 Berechnung der Multimenge $\mathcal{Q}$

Im Folgenden bezeichnet L_{T} die Menge der Transitivitätslemmata des aktuellen annotierten Programms, die durch das ✓eriFun-System bereits verifiziert sind. Weiter bezeichnet L die Menge aller verifizierten Lemmata lem mit $\mathit{lem} \notin L_{\mathrm{T}}$. Die Multimenge $\mathcal{Q}$ der A-Umgebungen der „*Computed Proof Rules*“ aus Abschnitt 14.3 zur Auswertung des Goal-Terms der Sequenz

$$H, IH \vdash g$$

wird mit Hilfe des Lemma-Filters wie folgt berechnet:

$$\begin{aligned}\mathcal{Q} := {} & \{1 * ih \mid ih \in IH\} \cup \\ & \{2 * q \mid (\texttt{lemma}\ \mathit{lem} \Leftarrow q) \in \mathit{filter}_F^*(L)\} \cup \\ & \{\mathit{depth}_{\max} * q \mid (\texttt{lemma}\ \mathit{lem} \Leftarrow q) \in L_{\mathrm{T}}\}\end{aligned}$$

Hierbei bezeichnet F die Menge der Prozeduren der Sequenz.

15.3 Wiederholte symbolische Auswertung

Betrachten wir das folgende Beispiel.

Beispiel 15.7. Sei P das Programm aus Beispiel 15.4. Wir betrachten diesmal die symbolische Auswertung der Sequenz

$$\begin{gathered}\{\neg\texttt{?0(k)}\}, \\ \{\texttt{ordered(qsort(larger(hd(k), tl(k))))}, \\ \texttt{ordered(qsort(smaller(hd(k), tl(k))))}\} \\ \vdash \\ \texttt{ordered(qsort(k))},\end{gathered} \tag{15.6}$$

Wir gehen wieder davon, dass die Gültigkeit der Lemmata aus Beispiel 15.4 bereits durch das System bewiesen wurde. Für die Sequenz (15.6) ergibt sich dann für die

$$
\begin{array}{lll}
\mathcal{U} \vdash & \texttt{ordered(qsort(k))} & \rightarrow \\
& \texttt{ordered(if(?empty(k),} & \\
& \quad \texttt{empty,} & \\
& \quad \texttt{ordered(app(qsort(smaller(hd(k), tl(k))),} & \rightarrow \\
& \qquad \texttt{add(hd(k), qsort(larger(hd(k), tl(k)))))))} & \\
& \texttt{ordered(if(false,} & \\
& \quad \texttt{empty,} & \\
& \quad \texttt{ordered(app(qsort(smaller(hd(k), tl(k))),} & \rightarrow \\
& \qquad \texttt{add(hd(k), qsort(larger(hd(k), tl(k)))))))} & \\
& \texttt{ordered(app(qsort(smaller(hd(k), tl(k))),} & \\
& \qquad \texttt{add(hd(k), qsort(larger(hd(k), tl(k))))))} &
\end{array}
$$

Abbildung 15.6: Symbolische Auswertung des Goal-Terms der Sequenz (15.6) aus Beispiel 15.7 unter Verwendung der Hypothese und der Induktionshypothesen.

Verwendung durch die „*Computed Proof Rules*“ die folgende Multimenge:

$$
\begin{array}{l}
\{\!\{1 * \texttt{ordered(qsort(larger(hd(k), tl(k))))}, \\
\quad 1 * \texttt{ordered(qsort(smaller(hd(k), tl(k))))}\}\!\}.
\end{array}
$$

Das bedeutet, dass wir zur symbolischen Auswertung des Goal-Terms von (15.6) keines der Lemmata aus Beispiel 15.4 verwenden. Die entsprechende symbolische Auswertung mit Hilfe der „*Simplification*“-Regel ist in Abbildung 15.6 dargestellt. Als Ergebnis der „*Simplification*“-Regel ergibt sich somit die Sequenz (15.5) aus Beispiel 15.4. Für diese Sequenz berechnet sich dann die folgende Multimenge für die „*Computed Proof Rules*“:

$$
\begin{array}{l}
\{\!\{1 * \texttt{ordered(qsort(larger(hd(k), tl(k))))}, \\
\quad 1 * \texttt{ordered(qsort(smaller(hd(k), tl(k))))}, \\
\quad 2 * \texttt{all n : nat, k : list[nat] if(lower\_bound(n, k), \ldots, true)}, \\
\quad 2 * \texttt{all n : nat, k : list[nat] if(upper\_bound(n, k), \ldots, true)}, \\
\quad 2 * \texttt{all n : nat, k : list[nat] lower\_bound(n, larger(n, k))}, \\
\quad 2 * \texttt{all n : nat, k : list[nat] upper\_bound(smaller(n, k), n)}, \\
\quad 2 * \texttt{all k, l : list[nat], n : nat if(ordered(k), \ldots, true)}\}\!\}.
\end{array}
\tag{15.7}
$$

Das bedeutet, dass wir zur symbolischen Auswertung des Goal-Terms der Sequenz (15.5) die Lemmata verwenden dürfen. Wie wir in Beispiel 15.4 bereits beschrieben haben, wertet die entsprechende symbolische Auswertung den Goal-Term der Sequenz zu `true` aus, womit wir insgesamt die Sequenz 15.6 verifiziert haben. Würden wir den Goal-Term der Sequenz (15.6) mit Hilfe der Multimenge (15.7) auswerten, so würde sich die Sequenz durch diese Auswertung direkt zu `true` vereinfachen. □

Wie das Beispiel zeigt, kann es aufgrund des Lemma-Filters notwendig sein, die „*Computed Proof Rules*“ wiederholt auf Sequenzen anzuwenden, um die entsprechenden Goal-Terme vollständig symbolisch auszuwerten. Diese wiederholte Anwendung der „*Computed Proof Rules*“ ist durchaus sinnvoll, da so jede der Sequenzen mit einer passenden Multimenge $\mathcal{Q}$ ausgewertet wird. Insgesamt ergeben sich dadurch effizientere symbolische Auswertungen, als wenn die Sequenzen ohne Berücksichtigung des Lemma-Filters mit Hilfe aller verifizierten Lemmata ausgewertet würden.

Es ist zu beachten, dass für die wiederholte Anwendung der „*Computed Proof Rules*“ keine Benutzerinteraktionen notwendig sind, da ✓`eriFun` mit Hilfe der „*Next-Rule-Heuristic*“ automatisch nach Anwendung einer „*Computed Proof Rule*“ wieder eine „*Computed Proof Rule*“ anwendet, sofern dies möglich ist.

Kapitel 16

Weitere Forschung und Schlussbemerkung

Wir haben in dieser Arbeit die vom ✓eriFun-System verwendete Programmiersprache sowie die Spezifikationssprache formal definiert und eine vollständige Beschreibung des symbolischen Auswertungskalküls angegeben. Hierbei haben wir insbesondere die Heuristiken zur Auswertung von Prozeduraufrufen und zur Anwendung von Klauseln ausführlich motiviert. Weiter haben wir den für das ✓eriFun-System entwickelten Lemma-Filter zur Ansteuerung der symbolischen Auswertung definiert. Damit stellt die vorliegende Arbeit eine vollständige Beschreibung der Beweiskomponenten des ✓eriFun-Systems dar und dient damit als Basis für alle zukünftigen Erweiterungen des Systems.

Wir wollen nun in diesem abschließenden Kapitel kurz den Einsatz des ✓eriFun-System am Fachgebiet Programmiermethodik der Technischen Universität Darmstadt beschreiben (Abschnitt 16.1) und Statistiken zu den berechneten Fallstudien angeben (Abschnitt 16.2). Anschließend gehen wir auf vergleichbare Arbeiten ein (Abschnitt 16.3) und stellen abschließend aktuelle Erweiterungen sowie mögliche zukünftige Erweiterungen des ✓eriFun-Systems vor (Abschnitt 16.4).

16.1 Einsatz in der Lehre und der Forschung

Das System wird in der halbjährlich stattfindenden Lehrveranstaltung „Praktikum Programmverifikation“ verwendet, um Studenten im Hauptstudium in die Programmverifikation einzuführen. In der ersten Praktikumsstunde wird den Studenten eine kurze Einführung in das ✓eriFun-System gegeben. Das System und die zugehörige Dokumentation [52] und [51] ist öffentlich über das Internet und damit auch für die Studenten verfügbar

Zunächst wird jede Woche per E-mail eine neue Verifikationsaufgabe an die Studenten verschickt. Diese Verifikationsaufgaben bestehen aus dem Nachweis der Korrektheit einer Reihe, den Studenten wohlbekannter, Sortierverfahren: *Insertion-Sort*, *Merge-Sort*, *Bubble-Sort*, *Minimum-Sort*, *Quick-Sort*, *Tree-Sort* und *Heap-Sort*. Hierbei steigt die Komplexität der Verifikationsaufgaben jede Woche. Insbesondere die abschließende Verifikation des *Heap-Sort*-Verfahrens ist nicht trivial.

Nachdem die Studenten diesen Teil des Praktikums erfolgreich abgeschlossen haben, werden sie in Gruppen eingeteilt und jede der Gruppen muss eine eigene nicht-triviale Verifikationsaufgabe lösen. Hierbei ist zu erwähnen, dass die gestellten Verifikationsaufgaben – im Gegensatz zur Verifikation der Sortierverfahren im ersten Teil des Praktikums – von den Veranstaltern des Praktikums nicht im Vor-

hinein bearbeitet und gelöst wurden. Die Studenten sind somit im Wesentlichen auf sich selbst gestellt und die Veranstalter können nur noch eine beratende Funktion einnehmen. Dieser Teil des Praktikums stellt deshalb für die Studenten eine große Herausforderung dar. Nichts desto trotz wurden seit der ersten Verwendung des ✓eriFun-Systems im Winter 2001/02 im Laufe der Zeit von den Studenten unter anderem die folgenden Verifikationsaufgaben erfolgreich gelöst:

- Unentscheidbarkeit des Halteproblems
- Korrektheit einer Implementierung von AVL-Bäumen
- Korrektheit des RSA-Verschlüsselungsverfahrens
- Korrektheit und Vollständigkeit eines einfachen Matching-Algorithmus
- Korrektheit und Vollständigkeit eines regelbasierten Unifikations-Algorithmus
- Korrektheit und Vollständigkeit des Robinsion-Unifikations-Algorithmus
- Korrektheit und Vollständigkeit eines „*recursive descent*“ Parsers
- Korrektheit des Boyer-Moore String-Search Verfahrens

Der Erfolg des Praktikums ist im Wesentlichen auf die einfache Logik des ✓eriFun-Systems und die benutzerfreundliche Oberfläche zurückzuführen. Sie ermöglichen den Studenten einen ausgesprochen schnellen Einstieg in das Arbeiten mit dem System. Die Studenten können sich von der „ersten Stunde“ an mit Verifikationsaufgaben beschäftigen und müssen nicht lange in das System eingeführt werden. Die geringe Anzahl der Beweisregeln des HPL-Kalküls (siehe Kapitel 2) ermöglicht es weiterhin, dass die Studenten den Beweiskalkül relativ schnell vollständig verstehen. Insbesondere können die Studenten die Beweisregeln durch einfaches Ausprobieren entsprechend Testen und so „spielerisch“ den Umgang mit dem System und den Beweisregeln lernen. Die Mächtigkeit der symbolischen Auswertung und damit die Mächtigkeit der „*Computed Proof Rules*“ des HPL-Kalküls reduzieren die Anzahl der Benutzerinteraktionen und verschaffen somit den Studenten ein schnelles Erfolgserlebnis. Die erfolgreiche Bearbeitung der abschließenden nicht-trivialen Beweisaufgaben durch die Studenten zeigt schließlich, dass das ✓eriFun-System sehr gut für eine praktische Einführung in die Programmverifikation geeignet ist. Ein ausführlicher Bericht über das Praktikum und die damit gesammelten Erfahrungen ist in [58] zu finden.

Neben dem Einsatz in der Lehre wurde ✓eriFun am Fachgebiet intern für Forschungsaufgaben eingesetzt. So wurden beispielsweise mit Hilfe des Systems die folgenden Fallstudien verifiziert:

- Korrektheit der binären Suche [54]
- Korrektheit des Fundamentalsatz der Arithmetik [53]
- Korrektheit eines Codegenerators für eine einfache imperative Sprache [55]

Durch den Einsatz sowohl in der Lehre als auch in der Forschung zeigt sich, dass die Einfachheit des ✓eriFun-Systems einen schnellen Einstieg in die Programmverifikation ermöglicht, ohne sich dabei auf triviale Verifikationsaufgaben beschränken zu müssen. Damit hat das ✓eriFun-System seine Zielsetzung erfolgreich erfüllt.

16.2 Statistiken der Fallstudien

Um den Automatisierungsgrad und damit die Mächtigkeit des ✓eriFun-Systems zu illustrieren, geben wir in den Abbildungen 16.1-16.8 Statistiken zu den bearbeiteten Fallstudien an. Für jede der in Abschnitt 16.1 genannten Fallstudie ist die Ausgabe des *Statistics Viewers* des ✓eriFun-Systems dargestellt. Um die Ausgabe des *Statistics Viewers* nachvollziehen zu können, beschreiben wir kurz die Bedeutung der einzelnen Tabellen:

- Die obere linke Tabelle des *Statistics Viewers* gibt einen Überblick darüber, wieviele Datenstrukturen, Prozeduren und Lemmata für eine Fallstudie formuliert werden mussten. Für Prozeduren bzw. Lemmata wird zwischen benutzerdefinierten Prozeduren bzw. Lemmata (*Verified Procedures* bzw. *Verified Lemmas*) und systemgenerierten Prozeduren bzw. Lemmata (*System Generated Procedures* bzw. *System Generated Lemmas*) unterschieden.

- Die obere rechte Tabelle gibt einen Überblick über die symbolischen Auswertungen der Fallstudie. Die erste Zeile (*GMA = Generated Main Nodes*) gibt die Summe der Längen der in der Fallstudie berechneten symbolischen Auswertungen an. Die zweite Zeile benennt das Lemma mit der längsten symbolischen Auswertung und gibt die Länge dieser symbolischen Auswertung an. Die dritte Zeile (*SRA = Successful Rule Applications*) gibt an, wieviele Regelanwendungen des überarbeiteten (Definition 14.2) und des erweiterten Auswertungskalküls (Definition 14.4) notwendig waren, um die symbolischen Auswertungen der Fallstudie zu berechnen. Die letzte Zeile gibt an, wieviel Zeit für die Berechnung der symbolischen Auswertungen benötigt wurde.

- Die Tabelle in der Mitte gibt einen Überblick über die angewendeten HPL-Regeln und ist somit zur Beurteilung des Automatisierungsgrads und der verwendeten Heuristiken des Systems besonders interessant. Für jede HPL-Regel wird angegeben, wie häufig sie angewendet wurde. Hierbei werden die Anwendungen in interaktive und automatische Anwendungen aufgeteilt. Es ist zu beachten, dass nur die interaktiven Regelanwendungen durch den Benutzer des Systems initiiert wurden. Die automatischen Anwendungen wurden durch die *Next-Rule-Heuristic* des ✓eriFun-Systems automatisch berechnet und bedurften keiner Benutzerinteraktion. Besonders interessant ist die letzte Spalte der letzten Zeile der Tabelle. Hier wird der Prozentsatz der automatischen Regelanwendungen angezeigt. Je höher dieser Wert ist desto besser ist der Automatisierungsgrad des ✓eriFun-Systems.

- Die untere Tabelle gibt einen Überblick über die Terminierungsbeweise der benutzerdefinierten Prozeduren. Da Terminierungsbeweise nicht Inhalt dieser Arbeit sind, ist diese Tabelle für uns uninteressant.

Die Fallstudien wurde mit der Version 3.1 des ✓eriFun-Systems, einer Weiterentwicklung des in dieser Arbeit beschrieben Systems, auf einem Pentium 4 mit einer Taktfrequenz von 2.2 GHz und 1 GB Hauptspeicher bearbeitet. Die Version 3.1 unterscheidet sich von der in dieser Arbeit beschriebenen Version im wesentlichen durch die folgenden Punkte:

- Der Auswertungskalkül wurde um Kürzungsregeln erweitert (siehe Abschnitt 16.4). Diese Kürzungsregeln ermöglichen zusätzliche symbolische Auswertungen.

- Die „*Execute*“-Regeln des Auswertungskalküls wurden so modifiziert, dass die Auswertung des Exception-Guards unnötig wird. Durch diese Änderungen wird der Auswertungskalkül geringfügig effizienter.

Die Fallstudien bzgl. des Halteproblems, der AVL-Bäume und des RSA-Verschlüsselungsverfahrens wurden mit der Version 2.5.6 des ✓eriFun-Systems bearbeitet und noch nicht auf die aktuelle Version des Systems übertragen. Aufgrund der erheblichen Modifikationen von Version 2.5.6 zu Version 3.1 sind die Statistiken der Version 2.5.6 nicht mehr aussagekräftig. Wir verzichten daher auf die Statistiken dieser Fallstudien.

16.3 Vergleiche zu anderen Arbeiten

Der wohl bekannteste und erfolgreichste Induktionsbeweiser ist ACL2 [27] (bzw. sein Vorgänger NQTHM [10]). Programme und Lemmata werden in ACL2 durch LISP-Ausdrücke spezifiziert. Nach der Spezifikation eines Lemmas kann der Benutzer ACL2 veranlassen, die Gültigkeit des Lemmas nachzuweisen. Einfluss auf den Nachweis der Gültigkeit hat der Benutzer nur durch die Definition so genannter *Hints*. Durch Hints kann der Benutzer festlegen, welche anderen Lemmata, Gleichungen, etc. zum Beweis des aktuellen Lemmas verwendet werden dürfen. Die Definition von Hints ist jedoch nur zu Beginn des Beweises des Lemmas möglich. Während des Beweises hat der Benutzer keine Einflussmöglichkeiten mehr. Schlägt der Beweis fehl, dann muss der Benutzer seine Hints modifizieren und den Beweis erneut starten. Zur Modifikation der Hints muss er die Ausgabe, des gescheiterten Beweises analysieren. Bei dieser Ausgabe handelt es sich um eine Art „handschriftlichen“ Beweis. Die Stärke von ACL2 liegt in der hohen Beweisautomatisierung. Viele Lemmata können vollautomatisch bewiesen werden. Nachteile von ACL2 sind die mangelnde Interaktion mit dem Benutzer und die hohe Komplexität. ACL2 versucht immer ein Lemma vollautomatisch zu beweisen. Schlägt dies fehl, so werden alle Zwischenergebnisse verworfen. Anschließend muss der Benutzer seine Hints modifizieren. Um dies erfolgreich zu tun, muss der Benutzer das ACL2-System sehr genau verstanden haben. Ein weiterer Nachteil von ACL2 ist die Ausgabe des Beweises in Form eines „handschriftlichen“ Beweises. Diese Ausgabe ist zur Beweisanalyse eines gescheiterten Beweises ausgesprochen unübersichtlich.

Neben ACL2 haben außerdem die taktikbasierten bzw. generischen Theorembeweiser wie Isabelle [35] und PVS [34] weite Verbreitung gefunden. Diese Theorembeweiser zeichnen sich insbesondere durch eine hohe Flexibilität aus. Basierend auf einigen wenigen Beweisregeln (*Basistaktiken*) wird dem Benutzer durch eine Steuersprache die Definition eigener *Taktiken* ermöglicht. Der Vorteil dieser Flexibilität ist, dass es dem Benutzer ermöglicht wird den Theorembeweiser für Themengebiete wie beispielsweise Zahlentheorie, Mengentheorie, Induktionsbeweise, etc. zu spezialisieren. Der Beweis eines Lemmas in einem taktikbasierten Theorembeweiser erfolgt typischerweise wie folgt: Auf die initiale Beweisverpflichtung wird durch den Benutzer eine Taktik angewendet. Diese Taktik erzeugt neue Beweisverpflichtungen, auf die der Benutzer wieder Taktiken anwenden kann. Dieses Interaktionsverhalten erhöht die Transparenz des Beweises enorm im Vergleich zu ACL2. Der Nachteil taktikbasierter Theorembeweiser ist, dass sie zunächst einmal eine geringe Beweisautomatisierung besitzen und somit viele Benutzerinteraktionen erforderlich sind. Dieser Nachteil wird dadurch reduziert, dass es für viele Themengebiete freiverfügbare Taktikbibliotheken gibt. Ein weiterer Nachteil taktikbasierter Theorembeweiser ist, dass die zugrunde liegende Logik komplex ist und somit eine entsprechend hohe Einarbeitungszeit vom Benutzer erforderlich ist.

✓eriFun verbindet das gute Interaktionsverhalten taktikbasierter Theorembeweiser mit der hohen Beweisautomatisierung von ACL2. Hierzu sind die Heuristiken zur Steuerung des symbolischen Auswerters von besonderer Bedeutung. Sie legen fest, wie weit ein Term ausgewertet wird und wann ein Term dem Benutzer zur erneuten Analyse zurückgeliefert wird. Außerdem ist die dem ✓eriFun-System zu-

Statistics Viewer "matching.vf"

Program Elements	Sum
Data Structures	7
Verified Procedures	14
System Generated Proce...	14
Verified Lemmas	21
System Generated Lemmas	0
Total	56

Symbolic Evaluation	
GMN	3055
"match is most-ge...	186
SRA	19829
SRA/GMN	6.5
SRA/sec	692.9
Time [hh:mm:ss]	0:00:28

Proof Rules	Total	Interactive	%	Automated	%
Simplification	106	0	0	106	100
Weak Simplification	0	0	0	0	0
Normalization	1	0	0	1	100
Weak Normalization	0	0	0	0	0
Inconsistency	0	-	-	0	0
Case Analysis	4	4	100	-	-
Use Lemma	6	2	33.3	4	66.7
Unfold Procedure	3	3	100	-	-
Apply Equation	5	2	40	3	60
Purge	0	0	0	0	0
Induction	22	1	4.5	21	95.5
Insert Induction Hypotheses	0	0	0	-	-
Insert Hypotheses	2	2	100	0	0
Move Hypotheses	2	0	0	2	100
Delete Hypotheses	1	0	0	1	100
Set Control	0	0	0	-	-
Total	152	14	9.2	138	90.8

Termination	Total	Interactive	%	Automated	%
Measure Terms	13	0	0	13	100

Close

Abbildung 16.1: Ausgabe des *Statistics Viewer* für die Fallstudie „Einfaches Matching“.

Statistics Viewer "Rule-Based_Unification.vf"

Program Elements	Sum
Data Structures	6
Verified Procedures	24
System Generated Procedures	27
Verified Lemmas	81
System Generated Lemmas	0
Total	138

Symbolic Evaluation	
GMN	27289
"((v• (σ ⊫ t))::σ ⊪ pl) = (...	815
SRA	147847
SRA/GMN	5.4
SRA/sec	474.3
Time [hh:mm:ss]	0:05:11

Proof Rules	Total	Interactive	%	Automated	%
Simplification	767	1	0.1	766	99.9
Weak Simplification	5	0	0	5	100
Normalization	14	1	7.1	13	92.9
Weak Normalization	1	1	100	0	0
Inconsistency	6	-	-	6	100
Case Analysis	2	2	100	-	-
Use Lemma	11	6	54.5	5	45.5
Unfold Procedure	12	12	100	-	-
Apply Equation	33	10	30.3	23	69.7
Purge	1	1	100	0	0
Induction	61	3	4.9	58	95.1
Insert Induction Hypotheses	0	0	0	-	-
Insert Hypotheses	0	0	0	0	0
Move Hypotheses	2	0	0	2	100
Delete Hypotheses	1	0	0	1	100
Total	916	37	4	879	96

Termination	Total	Interactive	%	Automated	%
Measure Terms	22	6	27.3	16	72.7

Close

Abbildung 16.2: Ausgabe des *Statistics Viewer* für die Fallstudie „Regelbasierte Unifikation“.

Statistics Viewer "Robinson_Unification.vf"

Program Elements	Sum
Data Structures	6
Verified Procedures	19
System Generated Procedures	21
Verified Lemmas	99
System Generated Lemmas	0
Total	145

Symbolic Evaluation	
GMN	20979
"ℜ"	402
SRA	118333
SRA/GMN	5.6
SRA/sec	71.4
Time [hh:mm:ss]	0:27:37

Proof Rules	Total	Interactive	%	Automated	%
Simplification	627	0	0	627	100
Weak Simplification	7	0	0	7	100
Normalization	22	3	13.6	19	86.4
Weak Normalization	7	7	100	0	0
Inconsistency	14	-	-	14	100
Case Analysis	2	2	100	-	-
Use Lemma	50	42	84	8	16
Unfold Procedure	10	10	100	-	-
Apply Equation	59	34	57.6	25	42.4
Purge	0	0	0	0	0
Induction	68	3	4.4	65	95.6
Insert Induction Hypotheses	0	0	0	-	-
Insert Hypotheses	4	4	100	0	0
Move Hypotheses	4	0	0	4	100
Delete Hypotheses	4	1	25	3	75
Total	878	106	12.1	772	87.9

Termination	Total	Interactive	%	Automated	%
Measure Terms	16	6	37.5	10	62.5

Close

Abbildung 16.3: Ausgabe des *Statistics Viewer* für die Fallstudie „Robinson-Unifikation“.

Statistics Viewer "rda.vf"

Program Elements	Sum
Data Structures	4
Verified Procedures	5
System Generated Proce...	5
Verified Lemmas	8
System Generated Lemmas	0
Total	22

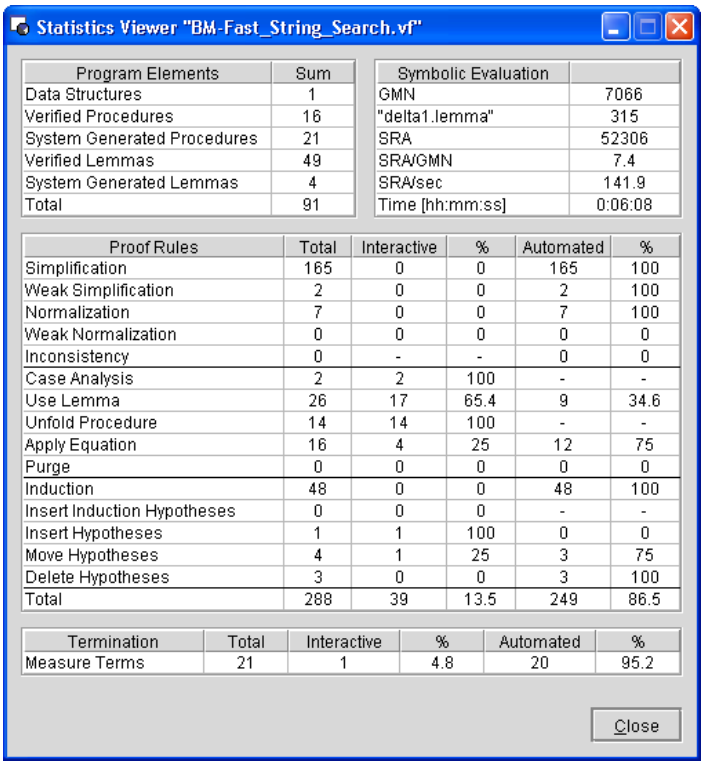
Symbolic Evaluation	
GMN	4456
"rda is complete s...	522
SRA	33179
SRA/GMN	7.4
SRA/sec	524.3
Time [hh:mm:ss]	0:01:03

Proof Rules	Total	Interactive	%	Automated	%
Simplification	63	0	0	63	100
Weak Simplification	0	0	0	0	0
Normalization	0	0	0	0	0
Weak Normalization	0	0	0	0	0
Inconsistency	0	-	-	0	0
Case Analysis	4	4	100	-	-
Use Lemma	15	14	93.3	1	6.7
Unfold Procedure	5	5	100	-	-
Apply Equation	3	0	0	3	100
Purge	0	0	0	0	0
Induction	7	0	0	7	100
Insert Induction Hypotheses	0	0	0	-	-
Insert Hypotheses	1	1	100	0	0
Move Hypotheses	0	0	0	0	0
Delete Hypotheses	0	0	0	0	0
Set Control	0	0	0	-	-
Total	98	24	24.5	74	75.5

Termination	Total	Interactive	%	Automated	%
Measure Terms	5	0	0	5	100

Close

Abbildung 16.4: Ausgabe des *Statistics Viewer* für die Fallstudie „Recursive Descent Parser“.



Statistics Viewer "BM-Fast_String_Search.vf"

Program Elements	Sum
Data Structures	1
Verified Procedures	16
System Generated Procedures	21
Verified Lemmas	49
System Generated Lemmas	4
Total	91

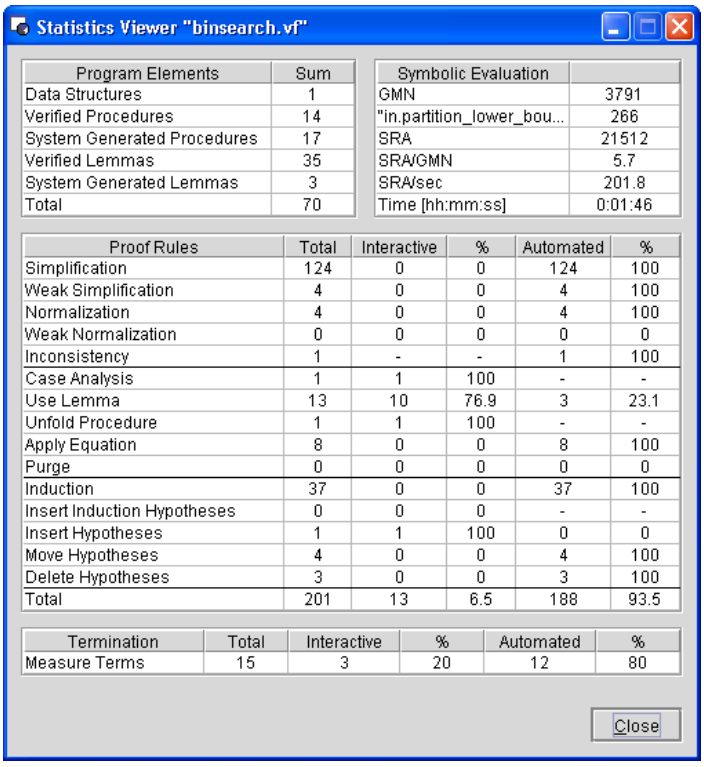
Symbolic Evaluation	
GMN	7066
"delta1.lemma"	315
SRA	52306
SRA/GMN	7.4
SRA/sec	141.9
Time [hh:mm:ss]	0:06:08

Proof Rules	Total	Interactive	%	Automated	%
Simplification	165	0	0	165	100
Weak Simplification	2	0	0	2	100
Normalization	7	0	0	7	100
Weak Normalization	0	0	0	0	0
Inconsistency	0	-	-	0	0
Case Analysis	2	2	100	-	-
Use Lemma	26	17	65.4	9	34.6
Unfold Procedure	14	14	100	-	-
Apply Equation	16	4	25	12	75
Purge	0	0	0	0	0
Induction	48	0	0	48	100
Insert Induction Hypotheses	0	0	0	-	-
Insert Hypotheses	1	1	100	0	0
Move Hypotheses	4	1	25	3	75
Delete Hypotheses	3	0	0	3	100
Total	288	39	13.5	249	86.5

Termination	Total	Interactive	%	Automated	%
Measure Terms	21	1	4.8	20	95.2

Close

Abbildung 16.5: Ausgabe des *Statistics Viewer* für die Fallstudie „Boyer-Moore String-Search“.



Statistics Viewer "binsearch.vf"

Program Elements	Sum
Data Structures	1
Verified Procedures	14
System Generated Procedures	17
Verified Lemmas	35
System Generated Lemmas	3
Total	70

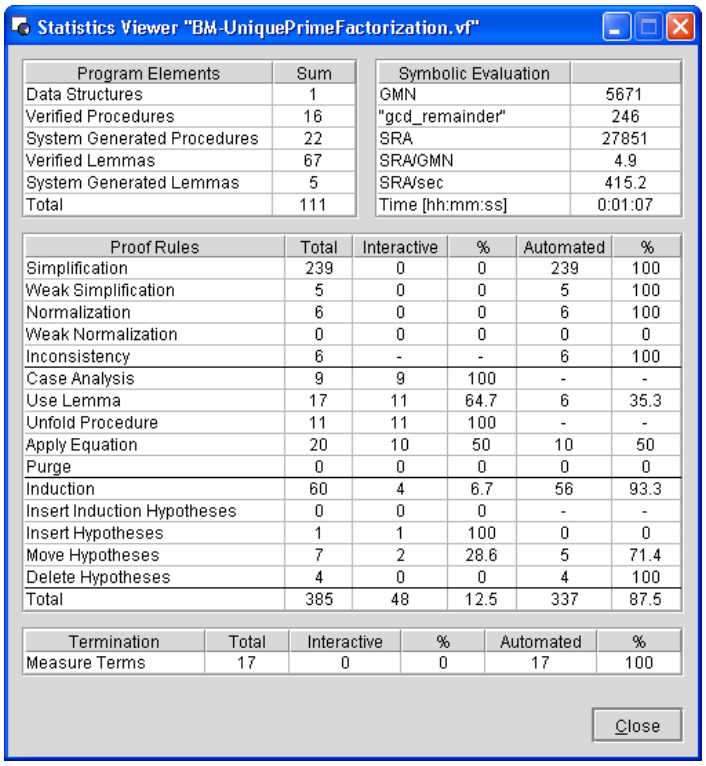
Symbolic Evaluation	
GMN	3791
"in.partition_lower_bou...	266
SRA	21512
SRA/GMN	5.7
SRA/sec	201.8
Time [hh:mm:ss]	0:01:46

Proof Rules	Total	Interactive	%	Automated	%
Simplification	124	0	0	124	100
Weak Simplification	4	0	0	4	100
Normalization	4	0	0	4	100
Weak Normalization	0	0	0	0	0
Inconsistency	1	-	-	1	100
Case Analysis	1	1	100	-	-
Use Lemma	13	10	76.9	3	23.1
Unfold Procedure	1	1	100	-	-
Apply Equation	8	0	0	8	100
Purge	0	0	0	0	0
Induction	37	0	0	37	100
Insert Induction Hypotheses	0	0	0	-	-
Insert Hypotheses	1	1	100	0	0
Move Hypotheses	4	0	0	4	100
Delete Hypotheses	3	0	0	3	100
Total	201	13	6.5	188	93.5

Termination	Total	Interactive	%	Automated	%
Measure Terms	15	3	20	12	80

Close

Abbildung 16.6: Ausgabe des *Statistics Viewer* für die Fallstudie „Binäre Suche“.



Statistics Viewer "BM-UniquePrimeFactorization.vf"

Program Elements	Sum
Data Structures	1
Verified Procedures	16
System Generated Procedures	22
Verified Lemmas	67
System Generated Lemmas	5
Total	111

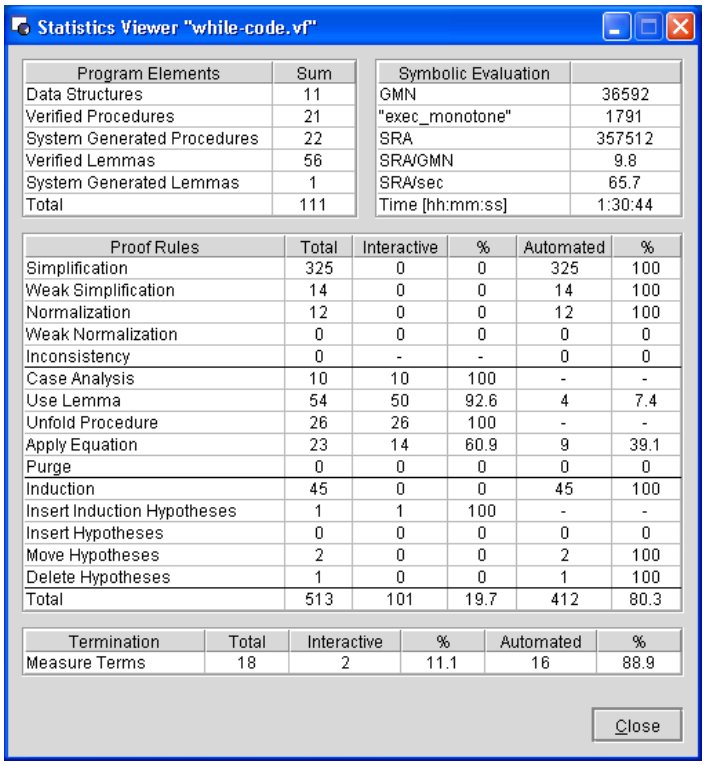
Symbolic Evaluation	
GMN	5671
"gcd_remainder"	246
SRA	27851
SRA/GMN	4.9
SRA/sec	415.2
Time [hh:mm:ss]	0:01:07

Proof Rules	Total	Interactive	%	Automated	%
Simplification	239	0	0	239	100
Weak Simplification	5	0	0	5	100
Normalization	6	0	0	6	100
Weak Normalization	0	0	0	0	0
Inconsistency	6	-	-	6	100
Case Analysis	9	9	100	-	-
Use Lemma	17	11	64.7	6	35.3
Unfold Procedure	11	11	100	-	-
Apply Equation	20	10	50	10	50
Purge	0	0	0	0	0
Induction	60	4	6.7	56	93.3
Insert Induction Hypotheses	0	0	0	-	-
Insert Hypotheses	1	1	100	0	0
Move Hypotheses	7	2	28.6	5	71.4
Delete Hypotheses	4	0	0	4	100
Total	385	48	12.5	337	87.5

Termination	Total	Interactive	%	Automated	%
Measure Terms	17	0	0	17	100

Close

Abbildung 16.7: Ausgabe des *Statistics Viewer* für die Fallstudie „Fundamentalsatz der Arithmetik“.



Statistics Viewer "while-code.vf"

Program Elements	Sum
Data Structures	11
Verified Procedures	21
System Generated Procedures	22
Verified Lemmas	56
System Generated Lemmas	1
Total	111

Symbolic Evaluation	
GMN	36592
"exec_monotone"	1791
SRA	357512
SRA/GMN	9.8
SRA/sec	65.7
Time [hh:mm:ss]	1:30:44

Proof Rules	Total	Interactive	%	Automated	%
Simplification	325	0	0	325	100
Weak Simplification	14	0	0	14	100
Normalization	12	0	0	12	100
Weak Normalization	0	0	0	0	0
Inconsistency	0	-	-	0	0
Case Analysis	10	10	100	-	-
Use Lemma	54	50	92.6	4	7.4
Unfold Procedure	26	26	100	-	-
Apply Equation	23	14	60.9	9	39.1
Purge	0	0	0	0	0
Induction	45	0	0	45	100
Insert Induction Hypotheses	1	1	100	-	-
Insert Hypotheses	0	0	0	0	0
Move Hypotheses	2	0	0	2	100
Delete Hypotheses	1	0	0	1	100
Total	513	101	19.7	412	80.3

Termination	Total	Interactive	%	Automated	%
Measure Terms	18	2	11.1	16	88.9

Close

Abbildung 16.8: Ausgabe des *Statistics Viewer* für die Fallstudie „Codegenerator für eine einfache imperative Sprache“.

grunde liegende Logik deutlich einfacher, als die Logiken von ACL2, Isabelle oder PVS. √eriFun basiert auf einer *first-order logic*, die um Regeln für Induktionsbeweise erweitert ist. ACL2, Isabelle und PVS basieren auf *higher-order logics*. Durch die einfachere Logik ist der Einstieg in √eriFun deutlich einfacher als in diese Systeme. Nachteil der einfacheren Logik ist, dass die Fallstudien, die aktuelle mit √eriFun bearbeitet werden können, noch nicht an die durch ACL2, Isabelle oder PVS bearbeiteten Fallstudien heranreichen.

Eine für Induktionsbeweise ausgesprochen erfolgreiche Heuristik ist das so genannte *Rippling* [13], [25]. Hier werden die Unterschiede zwischen Induktionskonklusion und Induktionshypothese explizit in der Induktionskonklusion markiert und nur dann Ersetzungsregeln angewendet, wenn der Unterschied zwischen Induktionskonklusion und Induktionshypothese aus der Konklusion rausgeschoben oder in eine frei Variable verschoben werden kann. Das Ziel von *Rippling* ist somit mit dem Ziel der Auswertungsregel „*Execute prodedure call (recursive cases)*“ unseres Auswertungskalküls vergleichbar, wobei *Rippling* nicht nur die Auswertungen von Prozeduren steuert, sondern generell die Anwendung von Ersetzungsregeln. Die Stärke von *Rippling* ist, dass es eine sehr zielgerichtete Methode ist und nur ein kleiner Suchraum für die Beweissuche abgearbeitet werden muss. *Rippling* wird unter anderem in den Beweissystemen INKA [9] und OYSTER/CLAM [12] verwendet.

16.4 Erweiterungen

Trotz des Erfolgs des √eriFun-Systems in der Lehre und der internen Verwendung am Fachgebiet hat das System natürlich auch seine Schwächen. Um einige dieser Schwächen zu beseitigen, sind zur Zeit die folgenden Erweiterungen des Systems in Bearbeitung:

(1) Da die Stärke und Eleganz des funktionalen Programmierens zu einem großen Teil durch die Verwendung von „*higher-order*“ Funktionen entsteht, ist es sinnvoll, die Programmiersprache $\mathcal{L}$ entsprechend zu erweitern. Diese Erweiterung der Sprache macht eine Modifikation des HPL-Kalküls, des Auswertungskalküls sowie der Terminierungsanalyse erforderlich. Die Modifikationen und ihre Implementierung sind in [5] beschrieben.

(2) Axiomatische Spezifikationen ermöglichen es, Funktionen axiomatisch zu beschreiben und dann Aussagen über diese Funktionen auf Basis der Axiome zu beweisen. Es können dann allgemeine Aussagen über Gruppen, Ringe, etc. formuliert und bewiesen werden. Die Definition und Implementierung axiomatischer Spezifikationen für √eriFun ist in [29] beschrieben.

(3) Für √eriFun existierte bisher lediglich ein binäres Speicherformat, welches insbesondere den Nachteil hat, dass die Dateien verschiedener √eriFun-Versionen inkompatibel sind. Es war daher bislang notwendig, bei Übertragung der Fallstudien auf eine neue Systemversion alle erforderlichen Benutzerinteraktionen erneut einzugeben. Mit einem auf Basis von OMDoc [28] zusätzlich implementierten Speicherformats [33] können jetzt Fallstudien automatisch bei Systemwechsel übertragen werden. Außerdem bildet dieses Speicherformat die Basis für den Beweisaustausch mit anderen Beweissystemen [33].

(4) Um den Benutzer bei der Beweisanalyse zu unterstützen, wurde ein Verfahren zur Generalisierung entwickelt und implementiert [1]. Durch dieses Verfahren wird auf Benutzeranfrage für eine Sequenz eine Generalisierung berechnet und diese Generalisierung in Form eines Lemmas in das System eingefügt und dann zu beweisen versucht.

(5) Die initial von einem Benutzer formulierten Lemmata sind, aufgrund fehlender Vorbedingungen häufig nicht korrekt. Um den Benutzer bei der Entdeckung solcher Fehler zu unterstützen, wurde ein Disprover für √eriFun entwickelt [44]. Dieser Disprover überprüft, nach Eingabe eines Lemmas, ob für das Lemma ein Gegenbeispiel konstruiert werden kann. Des weiteren soll der Disprover zur automatischen Überprüfung systemerzeugter Beweisverpflichtungen (Terminierungshypothesen, Formeln zur Rekursionselimination, Generalisierungen) eingesetzt werden.

(6) Das Typsystem der Programmiersprache $\mathcal{L}$ wird in [39] um Untertypen und abhängige Typen erweitert. Hierzu kann der Benutzer bei einer Prozedurdefinition Prädikate angeben, die die Argumente der Prozedur erfüllen müssen, damit ein entsprechender Prozeduraufruf wohlgetypt ist. Durch Untertypen und abhängige Typen vereinfacht sich sowohl die Verwendung unvollständigdefinierter Prozeduren durch den Benutzer als auch die Behandlung dieser Prozeduren durch den symbolischen Auswertungskalkül. Die Implementierung des erweiterten Typsystems wird in [20] beschrieben.

(7) Das System wurde zwischenzeitlich um Kürzungsregeln, die auch Funktionseigenschaften wie Assoziativität, Kommutativität und Idempotenz berücksichtigen, erweitert [18]. Beispielsweise wird ein Term

```
if(?0(y),
   true,
   plus(x,plus(y,times(y,z)))=plus(y,plus(times(x,y),x)))
```

zu $\texttt{if}(\texttt{?0}(\texttt{y}), \texttt{true}, \texttt{z} = \texttt{x})$ simplifiziert, wenn die Kürzungsregeln, Assoziativität und Kommutativität von `plus` und `times` zuvor verifiziert wurden.

Weiterhin sind die folgenden Erweiterungen des Systems angedacht:

(1) Axiomatische Spezifikationen werden häufig auf Basis von Gleichungen formuliert. Hierbei zeigt sich, dass die aktuelle Gleichheitsbehandlung durch den symbolischen Auswertungskalkül nicht ausreichend ist, um Beweise auf Basis axiomatischer Spezifikationen automatisch durchführen zu können. Es ist daher zu überlegen, ob zur Unterstützung der Gleichheitsbehandlung nicht ein Tool für Termersetzungssysteme in √eriFun effizient integriert werden kann.

(2) Es ist zu untersuchen, ob ein Compiler für die Programmiersprache $\mathcal{L}$ nicht eine sinnvolle Ergänzung für das √eriFun-System darstellt. Dadurch wäre es möglich, die verifizierten Programme auch wirklich zu verwenden. Bei der Implementierung eines solchen Compilers sollte auch untersuchen werden, inwieweit verifizierte Lemmata zur Optimierung des erzeugten Codes verwendet werden können.

(3) Die Benutzeroberfläche des √eriFun-Systems hat sich in der Anwendung des Systems bewährt. Die Anpassung und Weiterentwicklung der Benutzeroberfläche stellt jedoch einen nicht zu unterschätzenden Entwicklungsaufwand dar. Es ist daher sinnvoll darüber nachzudenken, die Oberfläche des √eriFun-System auf Basis eines Frameworks wie Eclipse [19] zu reimplementieren. Dies hätte den Vorteil, viele der durch den Framework gegebenen Features ohne großen Implementierungsaufwand nutzen zu können. Beispiele hierfür sind Projektmanagement, CVS-Integration, Syntax-Highlighting, Integration eines möglichen Compilers, etc.

Teil IV

Anhang

Anhang A

Mathematische Grundlagen

In diesem Anhang fassen wir einige der für diese Arbeit wichtigen mathematischen Begriffe und Notationen zusammen.

A.1 Mengen und Listen

Für Mengen $M_1, \ldots, M_n$ bezeichnen wir mit $M_1 \times \ldots \times M_n$ das endliche kartesische Produkt. Für $M_1 \times \ldots \times M_n$ schreiben wir auch kurz

$$\prod_{i=1}^{n} M_i.$$

Die Elemente des kartesischen Produkts notieren wir mit $\langle m_1, \ldots, m_m \rangle$, wobei gilt $m_i \in M_i$ für alle $i \in \{1, \ldots, n\}$. Für

$$\underbrace{M \times \ldots \times M}_{n\text{-mal}}$$

schreiben wir kurz M^n. Mit M^0 bezeichnen wir dann die Menge $\{\langle\rangle\}$. Das leere Tupel $\langle\rangle$ notieren wir gelegentlich auch durch ϵ. Für eine Familie von Mengen $(M_i)_{i \in I}$ mit unendlicher Indexmenge I bezeichnen wir mit

$$\prod_{i \in I} M_i$$

das unendliche kartesische Produkt. Die Elemente dieser Menge $\prod_{i \in I} M_i$ sind Funktionen $f : I \to \left(\bigcup_{i \in I} M_i\right)$ mit $f(i) \in M_i$ für alle $i \in I$. Für ein Element f von $\prod_{i \in I} M_i$ schreiben wir statt $f(i)$ auch $(f)_i$ und bezeichnen $(f)_i$ als i-te Komponente des Elements. Weiter bezeichnen wir mit $f[i/m]$ folgendes Element:

$$\begin{aligned} (f[i/m])_i &= m, \\ (f[i/m])_j &= (f)_j \quad \text{, falls } i \neq j. \end{aligned}$$

Für eine Menge M definieren wir die Menge M^* wie folgt:

$$M^* := \bigcup_{i=0}^{\infty} M^i.$$

Die Menge M^* enthält somit alle endlichen Tupel über der Menge M. Wir bezeichnen die Elemente der Menge M^* auch als endliche Listen und die Menge M^* als Menge der endlichen Listen über M. Die Menge der nicht-leeren, endlichen Listen

über M bezeichnen wir mit M^+. Für eine endliche Liste $L = \langle m_1, \ldots, m_n \rangle$ schreiben wir $m \in L$, falls ein $i \in \{1, \ldots, n\}$ existiert mit $m = m_i$. Weiter schreiben wir $L_1 \subseteq L_2$, falls für alle $m \in L_1$ auch $m \in L_2$ gilt. Wir definieren außerdem die folgenden Operationen auf endlichen Listen:

$$\begin{aligned}
m.\langle m_1, \ldots, m_n \rangle &:= \langle m, m_1, \ldots, m_n \rangle \\
\\
\langle m_1, \ldots, m_n \rangle \oplus \langle n_1, \ldots, n_k \rangle &:= \langle m_1, \ldots, m_n, n_1, \ldots, n_k \rangle \\
\\
\epsilon \setminus m &:= \epsilon \\
\langle m_1, m_2, \ldots, m_n \rangle \setminus m &:= \langle m_2, \ldots, m_n \rangle \setminus m \quad \text{falls } m_1 = m \\
\langle m_1, m_2, \ldots, m_n \rangle \setminus m &:= m_1.(\langle m_2, \ldots, m_n \rangle \setminus m) \quad \text{falls } m_1 \neq m \\
\epsilon \setminus_1 m &:= \epsilon \\
\langle m_1, m_2, \ldots, m_n \rangle \setminus_1 m &:= \langle m_2, \ldots, m_n \rangle \quad \text{falls } m_1 = m \\
\langle m_1, m_2, \ldots, m_n \rangle \setminus_1 m &:= m_1.(\langle m_2, \ldots, m_n \rangle \setminus_1 m) \quad \text{falls } m_1 \neq m
\end{aligned}$$

Für eine nicht-leere Menge M bezeichnen wir mit ε_0^M die Auswahlfunktion der nicht-leeren Teilmengen von M, d.h. es gilt

$$\varepsilon_0^M(N) \in N \quad \text{für alle } N \subseteq M \text{ mit } N \neq \emptyset.$$

Ist M aus dem Zusammenhang klar, so schreiben wir kurz ε_0. Die Existenz einer solchen Funktion für jede Menge M ist durch das Auswahlaxiom gewährleistet (siehe hierzu beispielsweise [16] oder [24]).

A.2 Relationen und partielle Funktionen

Eine reflexive, transitive und symmetrische Relation $\approx \subseteq M \times M$ bezeichnen wir als Äquivalenzrelation. Die Äquivalenzklassen einer Äquivalenzrelation $\approx$ definieren wir wie folgt:

$$[m]_\approx := \{m' \in M \mid m \approx m'\}.$$

Der kanonische Repräsentant einer Äquivalenzklasse $[m]_\approx$ ist dann wie folgt gegeben:

$$\varepsilon_\approx(m) := \varepsilon_0([m]_\approx).$$

Für eine Äquivalenzrelation $\approx \subseteq M \times M$ und eine Teilmengen N schreiben $a \in_\approx N$ falls ein $b \in N$ mit $a \approx b$ existiert. Weiter sind die folgenden Mengenoperatoren definiert:

- $N \cup_\approx N' := \{[m]_\approx \mid m \in N \cup N'\}$.
- $N \cap_\approx N' := \{[m]_\approx \mid \text{es existieren } m \in N \text{ und } m' \in N \text{ mit } m \approx m'\}$.
- $N \setminus_\approx N' := \{[m]_\approx \mid m \in N \text{ und für alle } m' \in N \text{ gilt } m \not\approx m'\}$.

Eine Relation $f \subseteq D \times E$ heißt partielle Funktion von D nach E gdw. für alle $d \in D$ höchstens ein $e \in E$ existiert mit $(d, e) \in f$. Statt $f \subseteq D \times E$ schreiben wir dann $f : D \mapsto E$ und statt $(d, e) \in f$ schreiben wir $f(d) = e$. Für eine partielle Abbildung $f : D \mapsto E$ bezeichnen wir die Menge

$$dom(f) = \{d \in D \mid \text{es existiert ein } e \in E \text{ mit } (d, e) \in f\}$$

als Domain. Eine partielle Funktion f mit $dom(f) = \{d_1, \ldots, d_n\}$ notieren wir mit $\{d_1 : f(d_1), \ldots, d_n : f(d_n)\}$. Für zwei partielle Abbildungen $f, g : D \mapsto E$ ist die partielle Abbildung $f \oplus g : D \mapsto E$ definiert als

$$(f \oplus g)(d) = \begin{cases} g(d) & \text{falls } d \in dom(g), \\ f(d) & \text{falls } d \in dom(f) \setminus dom(g). \end{cases}$$

Anhang B

Berechnung der Induktionsfälle

In Abschnitt 4.2 haben wir erwähnt, dass das ✓eriFun-System die durch Anwendung der HPL-Regel „*Induction*“ erzeugten Induktionsfälle mit Hilfe des Auswertungskalküls berechnet. Wir wollen in diesem Anhang diese Berechnung der Induktionsfälle beschreiben. Hierzu definieren wir zunächst in Abschnitt B.1 einige Operationen auf Listen und Mengen von Hypothesen. Anschließend definieren wir in Abschnitt B.2, wie Relationenbeschreibungen mit Hilfe des Auswertungskalküls vor der Generierung der Induktionsfälle vereinfacht werden. In den Abschnitten B.3 und B.4 beschreiben wir schließlich, wie die Induktionsschrittfälle und die Induktionsbasisfälle auf Basis der ausgewerteten Relationenbeschreibungen erzeugt werden.

B.1 Operationen für Hypothesen

Für eine Menge $M = \{h_1, \ldots, h_n\}$ von Literalen bzw. für eine Liste $L = \langle h_1, \ldots, h_n \rangle$ von Literalen und einen Term t bezeichnen wir mit $t{\downarrow_{P,M}}$ bzw. mit $t{\downarrow_{P,L}}$ den Term

$$\mathsf{e}((\mathsf{a}_{A_{\mathrm{std}}}(t)){\downarrow_{\mathcal{U}}}) \quad \text{mit} \quad \begin{array}{|ll|}\hline \mathcal{U}_0 & \mathrm{Hyp}_{\mathrm{G}} \\ \mathcal{U}: & \{h_1, \ldots, h_n\} \\ \hline \end{array}.$$

Während der Auswertung einer Relationenbeschreibung in Abschnitt B.2 ist es notwendig, eine Liste von Literalen mit Hilfe des Auswertungskalküls zu vereinfachen. Wir werten hierzu die Literale der Liste unter Berücksichtigung der anderen Literale aus.

Definition B.1. Sei P ein Programm und $L = \langle h_1, \ldots, h_n \rangle$ eine Liste von Literalen über P. Die Liste $L{\downarrow_P}$ ist dann definiert als $\langle h'_1, \ldots h'_n \rangle$, wobei für alle $i \in \{1, \ldots, n\}$ gilt

$$h'_i = h_i{\downarrow_{P,\{h'_1,\ldots,h'_{i-1},h_{i+1},\ldots,h_n\}}} \,.$$

□

Wir wollen die Fallunterscheidungen der Induktionsbasisfälle mit Hilfe positiver Strukturtests definieren, da sich so stärkere Hypothesenmengen für die Induktionsbasisfälle ergeben und so die Induktionsbasisfälle meist ohne zusätzliche Benutzerinteraktionen bewiesen werden können. Es ist daher notwendig, eine Hypothesenmenge mit negierten Strukturtests durch eine entsprechende Menge von Hypothesenmengen ohne negierte Strukturtests zu ersetzen. Wir führen hierzu den folgenden *Vervollständigungskalkül* ein.

Definition B.2 (Vervollständigungskalkül)**.** Sei P ein Programm. Der *Vervollständigungskalkül* ist dann wie folgt definiert:

(1) **Sprache:** Menge von Mengen von Literalen.

(2) **Inferenzregeln:**

$$\frac{\mathcal{H} \uplus \{H\}}{\mathcal{H}},$$

falls $?cons_i(t), ?cons_j(t)) \in H$ und $i \neq j$ gilt.

$$\frac{\mathcal{H} \uplus \{H\}}{\mathcal{H} \cup \{H_1, \ldots, H_{i-1}, H_{i+1}, \ldots, H_n\}},$$

falls $\neg ?cons_i(t) \in H$ und $H_j = (H \setminus \{\neg ?cons_i(t)\}) \cup \{?cons_j(t)\}$ für alle $j \in \{1, \ldots, n\} \setminus \{i\}$ gilt.[1]

(3) **Deduktion:** Wir schreiben $\mathcal{H} \rightharpoonup_P \mathcal{H}'$, falls eine Ableitung der Form

$$\frac{\mathcal{H}}{\mathcal{H}'}$$

existiert. Mit $\rightharpoonup_P^*$ bezeichnen wir die transitive-reflexive Hülle der Relation $\rightharpoonup_P$. Gilt $\mathcal{H} \rightharpoonup_P^* \mathcal{H}'$ und existiert kein $\mathcal{H}''$ mit $\mathcal{H}' \rightharpoonup_P \mathcal{H}''$, so nennen wir $\mathcal{H}'$ eine $\rightharpoonup_P$-Normalform von $\mathcal{H}$. Für jedes $\mathcal{H}$ existiert genau eine $\rightharpoonup_P$-Normalform. Wir bezeichnen diese $\rightharpoonup_P$-Normalform mit $\mathcal{H}\downharpoonright_P$. Statt $\{H\}\downharpoonright_P$ schreiben wir auch kurz $H\downharpoonright_P$.

□

Beispiel B.3. Sei P das Programm $\langle D_{\texttt{tree}}\rangle$, wobei $D_{\texttt{tree}}$ die Typoperatordefinition aus Abbildung 3.3 bezeichnet. Die Menge $\mathcal{H} = \{\{\neg ?\texttt{leaf}(t), \neg ?\texttt{nil}(t)\}\}$ können wir dann wie folgt vervollständigen:

$$\begin{array}{ll}
\{\{\neg ?\texttt{leaf}(\texttt{t}), \neg ?\texttt{nil}(\texttt{t})\}\} & \rightharpoonup_P \\
\{\{?\texttt{nil}(\texttt{t}), \neg ?\texttt{nil}(\texttt{t})\}, \{?\texttt{node}(\texttt{t}), \neg ?\texttt{nil}(\texttt{t})\}\} & \rightharpoonup_P \\
\{\{?\texttt{nil}(\texttt{t}), ?\texttt{leaf}(\texttt{t})\}, \{?\texttt{nil}(\texttt{t}), ?\texttt{node}(\texttt{t})\}, \{?\texttt{node}(\texttt{t}), \neg ?\texttt{nil}(\texttt{t})\}\} & \rightharpoonup_P \\
\{\{?\texttt{nil}(\texttt{t}), ?\texttt{node}(\texttt{t})\}, \{?\texttt{node}(\texttt{t}), \neg ?\texttt{nil}(\texttt{t})\}\} & \rightharpoonup_P \\
\{\{?\texttt{node}(\texttt{t}), \neg ?\texttt{nil}(\texttt{t})\}\} & \rightharpoonup_P \\
\{\{?\texttt{node}(\texttt{t}), ?\texttt{leaf}(\texttt{t})\}, \{?\texttt{node}(\texttt{t})\}\} & \rightharpoonup_P \\
\{\{?\texttt{node}(\texttt{t})\}\}. &
\end{array}$$

□

Für die Generierung der Induktionsschrittfälle verzichten wir auf eine Modifikation der Hypothesenmengen durch den Vervollständigungskalkül, da sich ansonsten zusätzliche Induktionsschrittfälle ergeben und sich so der Interaktionsaufwand des Benutzers erhöhen kann. Zur Berechnung der Hypothesenmengen der Induktionsschrittfälle verwenden wir vielmehr die folgende Funktion:

$$\begin{aligned}
?\text{-}norm_P(H) = \{h \in complete_P(H) \mid\ & h \neq \neg ?cons_i(t) \text{ oder} \\
& \text{es existiert kein Strukturprädikat} \\
& ?cons_j \text{ mit } ?cons_j(t) \in complete_P(H)\}.
\end{aligned}$$

Beispiel B.4. Sei P wieder das Programm $\langle D_{\texttt{tree}}\rangle$, wobei $D_{\texttt{tree}}$ die Typoperatordefinition aus Abbildung 3.3 bezeichnet. Es gilt dann:

$$\begin{aligned}
?\text{-}norm_P(\{\neg ?\texttt{nil}(\texttt{x}), ?\texttt{node}(\texttt{y})\}) &= \{\neg ?\texttt{nil}(\texttt{x}), ?\texttt{node}(\texttt{y})\} \\
?\text{-}norm_P(\{\neg ?\texttt{nil}(\texttt{x}), \neg ?\texttt{leaf}(\texttt{x}), ?\texttt{node}(\texttt{y})\}) &= \{?\texttt{node}(\texttt{x}), ?\texttt{node}(\texttt{y})\}.
\end{aligned}$$

□

[1] Wir gehen hierbei davon aus, dass $cons_1, \ldots, cons_n$ die Konstruktoren eines Typoperators bezeichnen.

Während der Berechnung der Hypothesenmengen der Induktionsfälle treten gelegentlich inkonsistente Hypothesenmengen auf. Für diese Hypothesenmengen müssen wir keine Induktionsfälle erzeugen, da diese Induktionsfälle trivialerweise gültig sind. Um konsistente bzw. inkonsistente Hypothesenmengen zu erkennen, definieren wir die das folgende Prädikat:

$$\mathit{consistent}_P(H) \quad :\Longleftrightarrow \quad \forall\, h \in H \text{ gilt } h{\downarrow_{P,H\setminus\{h\}}} \neq \texttt{false}.$$

B.2 Auswertung von Relationenbeschreibungen

Bevor wir eine Relationenbeschreibung zur Generierung der Induktionsfälle verwenden, wollen wir die Relationenbeschreibung mit Hilfe des Auswertungskalküls vereinfachen. Hierzu werten wir erstens die Domain-Terme der Relationenbeschreibungen und zweitens die Range-Terme der Range-Substitutionen aus. Für diese Auswertung der Relationenbeschreibung ist zu beachten, dass die symbolische Auswertung der Domain-Terme der Relationenbeschreibung unter Umständen neue Fallunterscheidungen einführen kann und so eine atomare Relationenbeschreibung in verschiedene atomare Relationenbeschreibungen aufgespalten werden muss. Zur Auswertung der Domain-Terme einer Relationenbeschreibung definieren wir daher die folgende Funktion:

$$\begin{aligned}\mathit{split\text{-}rel}_P(R) := \bigcup\nolimits_{\langle D,\Delta\rangle\in R}\{\langle D_1,\Delta\rangle,\ldots,\langle D_n,\Delta\rangle \mid \\ \langle D_1,\ldots,D_n\rangle = \mathit{DNF}(\mathit{AND}(D{\downarrow_P}))\}.\end{aligned}$$[2]

Die Range-Terme der Range-Substitutionen werden unter Berücksichtigung der jeweiligen Domain-Terme ausgewertet. Zur Generierung der Induktionsfälle auf Basis einer Relationenbeschreibung R verwenden wir dann die folgende ausgewertete Variante der Relationenbeschreibung R:

$$R{\downarrow_P} \quad := \quad \{\langle D, \Delta{\downarrow_{P,D}}\rangle \mid \langle D,\Delta\rangle \in \mathit{split\text{-}rel}_P(R)\}$$

Hierbei gilt $\Delta{\downarrow_{P,D}} := \{\{x_1/t_1{\downarrow_{P,D}},\ldots,x_n/t_n{\downarrow_{P,D}}\}\} \mid \{x_1/t_1,\ldots,x_n/t_n\} \in \Delta\}$.

B.3 Generierung der Induktionsschrittfälle

Wir betrachten im Folgenden die Generierung eines Induktionsschrittfalls für einen Goal-Term g mit $\mathit{fv}(g) = \{x_1,\ldots,x_n\}$ auf Basis einer atomaren Relationenbeschreibung $A = \langle D,\Delta\rangle$ mit $\Delta = \{\delta_1,\ldots,\delta_m\}$ sowie

$$\delta_i = \{x_{j_{1,i}}/t^i_1,\ldots,x_{j_{n_i,i}}/t^i_{n_i}\}.$$

Hierbei gehen wir davon aus, dass $y^1_1,\ldots,y^1_n,\ldots,y^m_1,\ldots,y^m_n$ als neue und paarweise verschiedene Termvariablen gegeben sind, so dass y^j_i und x_i für alle $i \in \{1,\ldots,n\}$ und alle $j \in \{1,\ldots,m\}$ den gleichen Typ haben. Die Menge der durch A definierten Induktionshypothesen ist dann gegeben durch

$$\mathit{IH}^A_g := \{\mathit{ih}_1,\ldots,\mathit{ih}_n\},$$

wobei gilt

$$\mathit{ih}_i = \texttt{all}\ y^i_1,\ldots,y^i_n\ \mathsf{e}((\sigma_i(\bar{\sigma}_i(\mathsf{a}_{A_{\mathrm{std}}}(g)))){\downarrow_{\mathcal{U}_D}}).$$

[2] Die Funktion *DNF* berechnet für einen boolschen Term b eine Liste von Listen von boolschen Termen. Diese Liste von Listen repräsentiert eine Disjunktiv-Normalform (DNF) des Terms b. Wie für einen boolschen Ausdruck eine DNF berechnet werden kann, kann in der einführenden Literatur zur formalen Logik nachgeschlagen werden (siehe beispielsweise [40]).

Die Termsubstitutionen σ_i und $\bar{\sigma}_i$ sind wie folgt gegeben:

$$\bar{\sigma}_i = \{x_1/y_1^i, \ldots, x_n/y_n^i\},$$

$$\sigma_i = \{y_{j_{1,i}}^i/\mathbf{a}_{A_{\text{top}}}(t_1^i), \ldots, y_{j_{n_i,i}}^i/\mathbf{a}_{A_{\text{top}}}(t_{n_i}^i)\}.$$

Die zur Vereinfachung der Induktionshypothesen verwendete A-Umgebung $\mathcal{U}_D$ mit $D = \langle h_1, \ldots, h_n\rangle$ ist wie folgt festgelegt:

$\mathcal{U}_0$	Proz.	Hyp_G
$\mathcal{U}_D$:	$\Sigma^{\text{proc}}(P)$	$\{h_1, \ldots, h_n\}$

Wir definieren dann die durch eine Relationenbeschreibung $R = \{\langle D_1, \Delta_1\rangle, \ldots, \langle D_n, \Delta_n\rangle\}$ definierten Induktionsschrittfälle wie folgt:

$$\begin{aligned} \textit{step-cases}_P^R(g) = \{H, IH_g^A \vdash g \mid\ & A = \langle\langle h_1, \ldots, h_n\rangle, \Delta\rangle \in R{\downarrow_P} \text{ und} \\ & (\texttt{all}\ y_1, \ldots, y_n\ \texttt{false}) \notin IH_g^A \text{ und} \\ & \textit{consistent}_P(\{h_1, \ldots, h_n\})) \text{ und} \\ & H = \texttt{?-}\textit{norm}_P(\{h_1, \ldots, h_n\})\} \end{aligned}$$

Die Bedingung $(\texttt{all}\ x_1', \ldots, x_n'\ \texttt{false}) \notin IH_g^A$ gewährleistet, dass keine Induktionsschrittfälle mit einer offensichtlich ungültigen Induktionshypothese erzeugt werden. Solche Induktionsschrittfälle sind trivialerweise gültig.

B.4 Generierung der Induktionsbasisfälle

Mit Hilfe der Induktionsschrittfälle definieren wir die Induktionsbasisfälle. Hierzu verknüpfen wir die Hypothesenmengen der Induktionsschrittfälle disjunktiv, negieren diese Disjunktion und generieren auf Basis dieses Terms

$$\neg(\mathit{OR}(\mathit{AND}(H_1), \ldots, \mathit{AND}(H_n)))$$

eine DNF. Damit haben wir die Fallunterscheidung für die Induktionsbasisfälle generiert. Hierbei ist zu beachten, dass wir die Hypothesenmengen der Induktionsschrittfälle zunächst mit Hilfe des Vervollständigungskalküls B.2 modifizieren, da sich so in der Regel kürzere DNFs ergeben.[3] Formal generieren wir also die Fallunterscheidung der Basisfälle mit Hilfe der folgenden Hypothesenmengen:

$$\begin{aligned} \mathcal{H}_\text{B} = \{\{h_1, \ldots, h_n\} \mid\ & \\ & \langle h_1, \ldots, h_n\rangle \in \mathit{DNF}(\neg(\mathit{OR}(\mathit{AND}(H_1{\downarrow_P}), \ldots, \mathit{AND}(H_n{\downarrow_P}))))\}{\downarrow_P}\,. \end{aligned}$$

Hierbei bezeichnen $H_1, \ldots, H_n$ die Hypothesenmengen der generierten Induktionsschrittfälle $\textit{step-cases}_P^R(g)$. Die Basisfälle sind dann wie folgt definiert:

$$\textit{step-bases}_P^R(g) = \{H, \emptyset \vdash g \mid H \in \mathcal{H}_\text{B} \text{ und } \textit{consistent}_P(H))\}.$$

[3] Der Term $\mathit{OR}(\mathit{AND}(H_1), \ldots, \mathit{AND}(H_n))$ ist eine Konjunktivnormalform (KNF). Das bedeutet, dass wir unsere DNF auf Basis einer KNF berechnen. Wie beispielsweise in [40] beschrieben ergeben sich in der Regel für längere Konjunktivnormalformen kürzere Disjunktivnormalformen. Da üblicherweise der Term $\mathit{OR}(\mathit{AND}(H_1{\downarrow_P}), \ldots, \mathit{AND}(H_n{\downarrow_P}))$ eine längere KNF bezeichnet als der Term $\mathit{OR}(\mathit{AND}(H_1), \ldots, \mathit{AND}(H_n))$, ergibt sich somit in der Regel für $\mathit{OR}(\mathit{AND}(H_1{\downarrow_P}), \ldots, \mathit{AND}(H_n{\downarrow_P}))$ eine kürzere DNF.

Anhang C

Definition von *perm-flat*$_{\mathcal{U},f}$

In diesem Anhang definieren wir die Funktion *perm-flat*$_{\mathcal{U},f}$, die für Terme $f(t^*)$ und $f(s^*)$ die Argumente $l_1, \ldots, l_m$ und $r_1, \ldots, r_m$ der gemeinsamen f-Termstruktur extrahiert und dann für diese Argumente eine Liste von Gleichungen

$$\langle l_1{=}r_1, \ldots, l_m{=}r_m \rangle$$

erzeugt, deren linke und rechte Seiten möglichst wenige syntaktische Unterschiede aufweisen. Hierbei bezeichnet $\mathcal{U}$ die aktuelle A-Umgebung der symbolischen Auswertung. Die Funktion wird in Kapitel 13 zur Definition der „*Functionality*"-Regeln verwendet.

Zur Definition der Funktion *perm-flat*$_{\mathcal{U},f}$ gehen wir wie folgt vor. In Abschnitt C.2 definieren wir zunächst eine Funktion *args*$_f$, die für die Terme $f(t^*)$ und $f(s^*)$ die Argumente der gemeinsamen f-Termstruktur ermittelt. Wir definieren dann in Abschnitt C.3 die Funktion *identical*$_{\mathcal{U}}$, die aus den Argumenten $l_1, \ldots, l_m$ und $r_1, \ldots, r_m$ die identischen Terme bzgl. $\simeq_{\mathcal{U}}$ herauszieht und für diese Terme entsprechende Gleichungen konstruiert. Für die restlichen Argumente definieren wir in Abschnitt C.4 die Funktion *similar*$_{\mathcal{U}}$. Diese Funktion berechnet für die Argumente Gleichungen mit möglichst ähnlichen Seiten. Aufbauend auf den Funktionen *args*$_f$, *identical*$_{\mathcal{U}}$ und *similar*$_{\mathcal{U}}$ definieren wir dann in Abschnitt C.5 die Funktion *perm-flat*$_{\mathcal{U},f}$.

Zum Vergleich von Termgleichungen auf Basis der Ähnlichkeit der linken und rechten Seite definieren wir jedoch zunächst in Abschnitt C.1 den so genannten *dm*-Kalkül und die Relation $>_{dm_P}$. Der *dm*-Kalkül ermittelt für zwei Terme l und r die Anzahl der syntaktischen Unterschiede und die Relation $>_{dm_P}$ verwendet dann diese Anzahl der syntaktischen Unterschiede um Termgleichungen $l{=}r$ entsprechend zu Vergleichen.

C.1 Difference Matching

Der *dm*-Kalkül ist auf Basis des in [8] vorgestellten „*difference matching*" wie folgt definiert:

Definition C.1 (*dm*-Kalkül)**.** Sei P ein Programm, $t^* = t_1, \ldots, t_n$ und $s^* = s_1, \ldots, s_m$. Der *dm-Kalkül* ist dann wie folgt gegeben:

(1) **Sprache:** Menge der Tupel der Form $\langle l \approx r, n \rangle$, wobei l und r Terme über P bezeichnen und n eine natürliche Zahl ist.

(2) **Inferenzregeln:**

$$\frac{-}{\langle x \approx x, 0\rangle}, \qquad \text{falls } x \in \mathcal{V}(\Omega(P))$$

$$\frac{-}{\langle t \approx t, 0\rangle}, \qquad \text{falls } t \in \mathcal{T}erm^{\text{cons}}(P)$$

$$\frac{\langle t_i \approx x), n\rangle}{\langle f(t^*) \approx x), (\#(f(t^*)) - \#(t_i)) + n\rangle}, \qquad \text{falls } x \in \mathcal{V}(\Omega(P)).$$

$$\frac{\langle x \approx s_i, n\rangle}{\langle x \approx g(s^*), (\#(g(s^*)) - \#(s_i)) + n\rangle}, \qquad \text{falls } x \in \mathcal{V}(\Omega(P))$$

$$\frac{\langle t_i \approx g(s^*), n\rangle}{\langle f(t^*) \approx g(s^*), (\#(f(t^*)) - \#(t_i)) + n\rangle}, \qquad \text{falls } f \neq g \text{ und } g(s^*) \notin \mathcal{T}erm^{\text{cons}}(P)^1$$

$$\frac{\langle f(t^*) \approx s_i, n\rangle}{\langle f(t^*) \approx g(s^*), (\#(g(s^*)) - \#(s_i)) + n\rangle}, \qquad \text{falls } f \neq g \text{ und } f(t^*) \notin \mathcal{T}erm^{\text{cons}}(P)$$

$$\frac{\langle t_1 \approx s_1, n_1\rangle \quad \ldots \quad \langle t_n \approx s_n, n_n\rangle}{\langle f(t^*) \approx f(s^*), n_1 + \ldots + n_n\rangle}, \qquad \text{falls } f(t^*), f(s^*) \notin \mathcal{T}erm^{\text{cons}}(P)$$

(3) **Deduktion:** Wir schreiben $\vdash^P_{dm} \langle l \approx r, n\rangle$, falls ein Ableitungsbaum der Form

$$\frac{\begin{array}{ccc}\vdots & & \vdots \\ \langle t_1 \approx s_1, n_1\rangle & \cdots & \langle t_m \approx s_m, n_m\rangle\end{array}}{\langle l \approx r, n\rangle}$$

existiert. Für zwei Terme l und r bezeichnen wir dann mit $dm_P(l, r)$ die kleinste natürliche Zahl n mit $\vdash^P_{dm} \langle l \approx r, n\rangle$. Existiert keine solche natürliche Zahl n, so so setzen wir $dm_P(l, r) = \mathit{undef}$.

□

Es ist zu beachten, dass Konstruktorgrundterme durch den Kalkül nicht zerlegt werden. Dies ist notwendig, da sich sonst ein zu hoher Aufwand zur Berechnung der Zahl $dm_P(l, r)$ ergeben kann. Betrachten wir zur Illustration des dm-Kalküls ein Beispiel:

Beispiel C.2. Für die Terme $\mathtt{plus(plus(x,y),plus(x,y))}$ und $\mathtt{plus(x,y)}$ gilt beispielsweise

$$\vdash^P_{\text{dm}} \langle \mathtt{plus(plus(x,y),plus(x,y))} \approx \mathtt{plus(x,y)}, 4\rangle,$$

da der folgende Ableitungsbaum existiert:

$$\frac{\dfrac{\dfrac{-}{\langle \mathtt{x} \approx \mathtt{x}, 0\rangle}}{\langle \mathtt{plus(x,y)} \approx \mathtt{x}, 2\rangle} \quad \dfrac{\dfrac{-}{\langle \mathtt{y} \approx \mathtt{y}, 0\rangle}}{\langle \mathtt{plus(x,y)} \approx \mathtt{y}, 2\rangle}}{\langle \mathtt{plus(plus(x,y),plus(x,y))} \approx \mathtt{plus(x,y)}, 4\rangle}$$

□

[1] Mit $\#(t)$ bezeichnen wir die Anzahl der Symbole im Term t.

Aufbauend auf dm_P definieren wir nun die Relation $>_{dm_P}$. Diese Relation ermittelt für zwei Gleichungen $l{=}r$ und $l'{=}r'$, welche der Gleichungen ähnlichere linke und rechte Seiten besitzt:

$$\left.\begin{array}{c} l{=}r >_{dm_P} l'{=}r' \\ \text{gdw.} \\ dm_P(l,r) \neq \mathit{undef} \text{ und } dm_P(l',r') \neq \mathit{undef} \text{ sowie } A < A' \text{ mit} \\ \begin{array}{rcl} A & = & \#(l) + \#(r) - dm_P(l,r), \\ A' & = & \#(l') + \#(r') - dm_P(l',r'). \end{array} \end{array}\right\} \quad \text{(C.1)}$$

Der Quotient $\frac{A}{2}$ bzw. $\frac{A'}{2}$ repräsentiert die Anzahl der Übereinstimmungen pro Seite der jeweiligen Gleichung und der Quotient $\frac{A}{\#(l)+\#(r)}$ bzw. $\frac{A'}{\#(l')+\#(r')}$ repräsentiert die Anzahl der Übereinstimmungen pro Symbol der jeweiligen Gleichung. Gilt $l{=}r >_{dm_P} l'{=}r'$, so bezeichnen wir die Terme l' und r' als ähnlicher als die Terme l und r.

C.2 Extrahieren der Argumente

Zum Extrahieren der Argumente $l_1, \ldots, l_m$ und $r_1, \ldots, r_m$ aus der gemeinsamen f-Termstruktur von $f(t^*)$ und $f(s^*)$ definieren wir die folgende Funktion:

$$\mathit{args}_f(f(t^*), f(s^*)) \quad := \quad \mathit{split}(\mathit{flat}_f(f(t^*), f(s^*))).$$

Die Funktion *split* ist dabei wie folgt gegeben:

$$\mathit{split}(\langle l_1{=}r_1, \ldots, l_n{=}r_n\rangle) \quad := \quad \langle\langle l_1, \ldots, l_n\rangle, \langle r_1, \ldots, r_n\rangle\rangle.$$

C.3 Gleichungen mit identischen Termen

Nachdem wir nun die Argumente $l_1, \ldots, l_m$ und $r_1, \ldots, r_m$ der gemeinsamen f-Termstruktur ermittelt haben, erzeugen wir für diese Argumente Gleichungen mit identischen linken und rechten Seiten. Hierzu führen wir zunächst die folgende Funktion ein:

$$\begin{array}{rcll} \mathit{id\text{-}list}_{\mathcal{U}}(\epsilon, R) & := & \langle \epsilon, \epsilon, R\rangle, & \\ \mathit{id\text{-}list}_{\mathcal{U}}(l.L, R) & := & \langle l{=}l.E', L', R'\rangle, & \text{falls } l \in_{\simeq_{\mathcal{U}}} R \text{ und} \\ & & & \langle E', L', R'\rangle = \mathit{id\text{-}list}_{\mathcal{U}}(L, R \setminus_{1,\simeq_{\mathcal{U}}} \{l\}),^2 \\ \mathit{id\text{-}list}_{\mathcal{U}}(l.L, R) & := & \langle E', l.L', R'\rangle, & \text{falls } l \notin_{\simeq_{\mathcal{U}}} R \text{ und} \\ & & & \langle E', L', R'\rangle = \mathit{id\text{-}list}_{\mathcal{U}}(L, R). \end{array}$$

Die Funktion zur Berechnung der Gleichungen mit identischen linken und rechten Seiten ist dann wie folgt definiert:

$$\mathit{identical}_{\mathcal{U}}(f(t^*), g(s^*)) \quad := \quad E'$$

$$\text{mit } \langle E', L', R'\rangle = \mathit{id\text{-}list}_{\mathcal{U}}(L, R) \text{ und } \langle L, R\rangle = \mathit{args}_f(f(t^*), g(s^*)).$$

[2] Für eine Liste von Termen $D = \langle t_1, \ldots, t_n\rangle$ und einen Term t bezeichnen wir mit $D \setminus_{1,\simeq_{\mathcal{U}}} t$ die Liste $\langle t_1, \ldots, t_{i-1}, t_{i+1}, \ldots, t_n\rangle$, wobei i den kleinste Index mit $t_i \simeq_{\mathcal{U}} t$ bezeichnet.

C.4 Gleichungen mit ähnlichen Termen

Zur Generierung der Gleichungen mit möglichst ähnlichen linken und rechten Seiten definieren wir zunächst die Funktion $sim_{\mathcal{U}}$. Diese Funktion ermittelt für einen Term l und eine Liste von Termen R den Term $r \in R$, für den die Gleichung $l{=}r$ bzgl. der Relation $>_{dm_P}$ einen minimalen Unterschied aufweist. Die Funktion $sim_{\mathcal{U}}$ ist wie folgt gegeben:

$$sim_{\mathcal{U}}(l, R) \quad := \quad r$$

falls $dm_P(l{=}r) \neq \mathit{undef}$ und $\forall\, r' \in R$ entweder $dm_P(l{=}r') = \mathit{undef}$ oder $l{=}r' >_{dm_P} l{=}r$ gilt.

Gilt $dm_P(l{=}r) = \mathit{undef}$ für alle $r \in R$, so setzen wir $sim_{\mathcal{U}}(l, R) = \mathit{undef}$. Mit Hilfe der Funktion $sim_{\mathcal{U}}(l, R)$ definieren wir dann die Funktion $\mathit{sim\text{-}list}_{\mathcal{U}}$. Diese Funktion generiert für zwei Listen von Termen eine Liste von Gleichungen mit möglichst ähnlichen linken und rechten Seiten. Hierzu ermittelt die Funktion für den ersten Term aus L den ähnlichsten Term r aus R. Für den zweiten Term aus L betrachten wir dann nur noch die um r reduzierte Liste usw. Die Funktion ist wie folgt definiert:

$$\mathit{sim\text{-}list}_{\mathcal{U}}(\epsilon, \epsilon) \quad := \quad \epsilon,$$

$$\mathit{sim\text{-}list}_{\mathcal{U}}(l.L, R) \quad := \quad \begin{cases} l{=}r.E, & \text{falls } r = sim_{\mathcal{U}}(l, R) \text{ mit } r \neq \mathit{undef} \text{ und} \\ & E = \mathit{sim\text{-}list}_{\mathcal{U}}(l.L, R \backslash r) \text{ mit } E \neq \mathit{undef}, \\ \mathit{undef}, & \text{sonst.} \end{cases}$$

Auf Basis dieser Funktion definieren wir dann schließlich die Funktion $similar_{\mathcal{U}}$:

$$similar_{\mathcal{U}}(f(t^*), g(s^*)) \quad := \quad \mathit{sim\text{-}list}_{\mathcal{U}}(L', R')$$

mit $\langle E', L', R' \rangle = \mathit{id\text{-}list}_{\mathcal{U}}(L, R)$ und $\langle L, R \rangle = args_f(f(t^*), g(s^*))$.

C.5 Definition von $\mathit{perm\text{-}flat}_{\mathcal{U},f}$

Die Funktion $\mathit{perm\text{-}flat}_{\mathcal{U},f}$ ist aufbauend auf den Funktionen $identical_{\mathcal{U}}$ und $similar_{\mathcal{U}}$ wie folgt definiert:

$$\mathit{perm\text{-}flat}_{\mathcal{U},f}(t, r) \quad := \quad \begin{cases} I \oplus S, & \text{falls } I \neq \mathit{undef} \text{ und } S \neq \mathit{undef}, \\ \mathit{flat}_P(t, r), & \text{sonst.} \end{cases}$$

Hierbei ist $I = identical_{\mathcal{U}}(t, r)$ und $S = similar_{\mathcal{U}}(t, r)$.

Anhang D

„*Unfold*“-Regeln für kommutative Prozeduren

Wir definieren in diesem Anhang für jede „*Unfold*“-Regel aus Kapitel 10 eine entsprechende kommutierte Version.

D.1 „*Unfold procedure call**“

54. Unfold procedure call*

$$\frac{f^{\,A}(t_1, t_2)}{\mathsf{env}_{\mathcal{U}'}(\sigma_{\xi_{f(\ldots)}}(\mathsf{s}^*_f(\mathsf{dec}_f(\mathsf{un}_f^{u-1,0,m}(R_f)))))}, \qquad \begin{array}{|c|cc|}\hline & \text{Label} & \text{U-Lim.} \\ \hline A\text{:} & l & u \\ \hline \end{array}$$

falls $u > 0$, $f \in \Sigma \cap \Sigma_{\mathrm{C}} \cap \Sigma^{\mathrm{rec}}(P)$, $l \notin C$, $\sigma = \{x_1/\mathsf{a}_{A_0}(t_2), x_2/\mathsf{a}_{A_0}(t_1)\}$, $m = mark(\sigma)$ und $f \in \bigcup_{h \in H_{\mathrm{L}} \cup H_{\mathrm{G}}} proc_P(h)$. Weiter muss folgendes gelten:

$$\left.\begin{array}{l}(\sigma_{\xi_{f(\ldots)}}(\mathsf{s}^*_f(\mathsf{dec}_f(\mathsf{un}_f^{u-1,0,m}(R_f))))) \downarrow_{\mathcal{U}'} \text{ enthält keinen Teil-} \\ \text{term der Form } f^{\langle \ldots, m, \ldots \rangle}(\ldots).\end{array}\right\} (*)$$

Hierbei bezeichnet $\mathcal{U}'$ die folgende A-Umgebung:

	Reg.
$\mathcal{U}'$:	$no\text{-}execute(\mathcal{R})$

D.2 „*Unfold structure predicate argument**“

64. Unfold structure predicate argument*

$$\frac{?cons(f^{\boxed{A}}(t_1,t_2))}{\mathsf{env}_{\mathcal{U}'}(?cons^{\boxed{A_{\mathrm{std}}}}(\sigma_{\xi_{f(\ldots)}}(\mathsf{s}^*_f(\mathsf{dec}_f(\mathsf{un}_f^{u-1,0,m}(R_f))))))},$$

	Label	U-Lim.
A:	l	u

falls $u > 0$, $f \in (\Sigma \cap \Sigma_{\mathrm{C}}) \setminus \Sigma^{\mathrm{tail}}(P)$, $l \notin C$, $\sigma = \{x_1/\mathsf{a}_{A_0}(t_2), x_2/\mathsf{a}_{A_0}(t_1)\}$, $m = mark(\sigma)$ und folgendes gilt:

$$\left.\begin{array}{l}(?cons^{\boxed{A_{\mathrm{std}}}}(\sigma_{\xi_{f(\ldots)}}(\mathsf{s}^*_f(\mathsf{dec}_f(\mathsf{un}_f^{u-1,0,m}(R_f)))))) \downarrow_{\mathcal{U}'} \text{ enthält} \\ \text{keinen Teilterm der Form } f^{\boxed{\langle\ldots,m,\ldots\rangle}}(\ldots).\end{array}\right\}(*)$$

Hierbei bezeichnet $\mathcal{U}'$ die folgende A-Umgebung:

	Reg.
$\mathcal{U}'$:	$no\text{-}execute(\mathcal{R})$

D.3 „*Unfold selector argument**“

66. Unfold selector argument*

$$\frac{sel(f^{\boxed{A}}(t_1,t_2))}{\mathsf{env}_{\mathcal{U}'}(sel^{\boxed{A_{\mathrm{std}}}}(\sigma_{\xi_{f(\ldots)}}(\mathsf{s}^*_f(\mathsf{dec}_f(\mathsf{un}_f^{u-1,0,m}(R_f))))))},$$

	Label	U-Lim.
A:	l	u

falls $u > 0$, $f \in (\Sigma \cap \Sigma_{\mathrm{C}}) \setminus \Sigma^{\mathrm{tail}}(P)$, $l \notin C$, $\sigma = \{x_1/\mathsf{a}_{A_0}(t_2), x_2/\mathsf{a}_{A_0}(t_1)\}$, $m = mark(\sigma)$ und folgendes gilt:

$$\left.\begin{array}{l}(sel^{\boxed{A_{\mathrm{std}}}}(\sigma_{\xi_{f(\ldots)}}(\mathsf{s}^*_f(\mathsf{dec}_f(\mathsf{un}_f^{u-1,0,m}(R_f)))))) \downarrow_{\mathcal{U}'} \text{ enthält kei-} \\ \text{nen Teilterm der Form } f^{\boxed{\langle\ldots,m,\ldots\rangle}}(\ldots).\end{array}\right\}(*)$$

Hierbei bezeichnet $\mathcal{U}'$ die folgende A-Umgebung:

	Reg.
$\mathcal{U}'$:	$no\text{-}execute(\mathcal{R})$

D.4 „*Unfold case condition**“

68. Unfold case condition*

$$\frac{\mathsf{case}(f^{\,A}(t_1,t_2),cons^* : t^*)}{\mathsf{env}_{\mathcal{U}'}(\mathsf{case}^{\,A_{\mathrm{std}}}(\sigma_{\xi_{f(\ldots)}}(\mathsf{s}^*_f(\mathsf{dec}_f(\mathsf{un}^{u-1,0,m}_f(R_f)))),cons^* : \mathsf{a}_{A_0}(t^*)))},$$

falls $u > 0$, $f \in \Sigma \cap \Sigma_{\mathrm{C}} \cap \Sigma^{\mathrm{rec}}(P)$, $l \notin C$, $\sigma = \{x_1/\mathsf{a}_{A_0}(t_2), x_2/\mathsf{a}_{A_0}(t_1)\}$, $m = mark(\sigma)$ und folgendes gilt:

$$\left.\begin{array}{l}(\mathsf{case}^{\,A_{\mathrm{std}}}(\sigma_{\xi_{f(\ldots)}}(\mathsf{s}^*_f(\mathsf{dec}_f(\mathsf{un}^{u-1,0,m}_f(R_f)))),cons^* : \mathsf{a}_{A_0}(t^*)))\downarrow_{\mathcal{U}'}\\ \text{enthält keinen Teilterm der Form } f^{\,\langle\ldots,m,\ldots\rangle}(\ldots).\end{array}\right\}(*)$$

Hierbei bezeichnet $\mathcal{U}'$ die folgende A-Umgebung und A die folgende A-Annotation:

	Reg.
$\mathcal{U}'$:	$no\text{-}execute(\mathcal{R})$

	Lab.	U.
A:	l	u

D.5 „*Unfold left equality argument**“

70. Unfold left equality argument*

$$\frac{f^{\,A}(t_1,t_2){=}r}{\mathsf{env}_{\mathcal{U}'}(\sigma_{\xi_{f(\ldots)}}(\mathsf{s}^*_f(\mathsf{dec}_f(\mathsf{un}^{u-1,0,m}_f(R_f)))){=}^{\,A_{\mathrm{std}}}\mathsf{a}_{A_0}(r)},$$

	Label	U-Lim.
A:	l	u

falls $u > 0$, $f \in \Sigma \cap \Sigma_{\mathrm{C}} \cap \Sigma^{\mathrm{rec}}(P)$, $l \notin C$, $\sigma = \{x_1/\mathsf{a}_{A_0}(t_2), x_2/\mathsf{a}_{A_0}(t_1)\}$, $m = mark(\sigma)$ und folgendes gilt:

$$\left.\begin{array}{l}(\sigma_{\xi_{f(\ldots)}}(\mathsf{s}^*_f(\mathsf{dec}_f(\mathsf{un}^{u-1,0,m}_f(R_f)))){=}^{\,A_{\mathrm{std}}}\mathsf{a}_{A_0}(r))\downarrow_{\mathcal{U}'}\text{ enthält keinen}\\ \text{Teilterm der Form } f^{\,\langle\ldots,m,\ldots\rangle}(\ldots).\end{array}\right\}(*)$$

Hierbei bezeichnet $\mathcal{U}'$ die folgende A-Umgebung:

	Reg.
$\mathcal{U}'$:	$no\text{-}execute(\mathcal{R})$

D.6 „*Unfold right equality argument**“

72. Unfold right equality argument*

$$\frac{l{=}f^{\,A}(t_1,t_2)}{\mathsf{env}_{\mathcal{U}'}(\mathsf{a}_{A_0}(l){=}^{\,A_{\mathrm{std}}}\sigma_{\xi_{f(\ldots)}}(\mathsf{s}^*_f(\mathsf{dec}_f(\mathsf{un}^{u-1,0,m}_f(R_f)))))},$$

	Label	U-Lim.
A:	l	u

falls $u > 0$, $f \in \Sigma \cap \Sigma_{\mathrm{C}} \cap \Sigma^{\mathrm{rec}}(P)$, $l \notin C$, $\sigma = \{x_1/\mathsf{a}_{A_0}(t_2), x_2/\mathsf{a}_{A_0}(t_1)\}$, $m = mark(\sigma)$ und folgendes gilt:

$$\left.\begin{array}{l}(\mathsf{a}_{A_0}(l){=}^{\,A_{\mathrm{std}}}\sigma_{\xi_{f(\ldots)}}(\mathsf{s}^*_f(\mathsf{dec}_f(\mathsf{un}^{u-1,0,m}_f(R_f))))) \downarrow_{\mathcal{U}'} \text{ enthält keinen}\\ \text{Teilterm der Form } f^{\,\langle\ldots,m,\ldots\rangle}(\ldots).\end{array}\right\}(*)$$

Hierbei bezeichnet $\mathcal{U}'$ die folgende A-Umgebung:

	Reg.
$\mathcal{U}'$:	$no\text{-}execute(\mathcal{R})$

D.7 „*Unfold structure predicate selector argument**“

74. Unfold structure predicate selector argument*

$$\frac{?cons(sel(f^{\,A}(t_1,t_2)))}{\mathsf{env}_{\mathcal{U}'}(?cons^{\,A_{\mathrm{std}}}(sel^{\,A_{\mathrm{std}}}(\sigma_{\xi_{f(\ldots)}}(\mathsf{s}^*_f(\mathsf{un}^{u-1,0,m}_f(R_f))))))},$$

	Label	U-Lim.
A:	l	u

falls $u > 0$, $f \in (\Sigma \cap \Sigma_{\mathrm{C}}) \setminus \Sigma^{\mathrm{tail}}(P)$, $l \notin C$, $\sigma = \{x_1/\mathsf{a}_{A_0}(t_2), x_2/\mathsf{a}_{A_0}(t_1)\}$, $m = mark(\sigma)$ und folgendes gilt:

$$\left.\begin{array}{l}(?cons^{\,A_{\mathrm{std}}}(sel^{\,A_{\mathrm{std}}}(\sigma_{\xi_{f(\ldots)}}(\mathsf{s}^*_f(\mathsf{un}^{u-1,0,m}_f(R_f)))))) \downarrow_{\mathcal{U}'} \text{ enthält kei-}\\ \text{nen Teilterm der Form } f^{\,\langle\ldots,m,\ldots\rangle}(\ldots).\end{array}\right\}(*)$$

Hierbei bezeichnet $\mathcal{U}'$ die folgende A-Umgebung:

	Reg.
$\mathcal{U}'$:	$no\text{-}execute(\mathcal{R})$

D.8 „*Unfold left equality selector argument**“

76. Unfold left equality selector argument*

$$\frac{sel(f^{\,A}(t_1,t_2))=r}{\mathbf{env}_{\mathcal{U}'}(sel^{\,A_{\mathrm{std}}}(\sigma_{\xi_{f(\ldots)}}(\mathbf{s}^*_f(\mathbf{un}^{u-1,0,m}_f(R_f))))=^{\,A_{\mathrm{std}}}\mathbf{a}_{A_0}(r))},$$

	Label	U-Lim.
A:	l	u

falls $u > 0$, $f \in (\Sigma \cap \Sigma_{\mathrm{C}}) \setminus \Sigma^{\mathrm{tail}}(P)$, $l \notin C$, $\sigma = \{x_1/\mathbf{a}_{A_0}(t_2), x_2/\mathbf{a}_{A_0}(t_1)\}$, $m = mark(\sigma)$ und folgendes gilt:

$$\left.\begin{array}{l}(sel^{\,A_{\mathrm{std}}}(\sigma_{\xi_{f(\ldots)}}(\mathbf{s}^*_f(\mathbf{un}^{u-1,0,m}_f(R_f))))=^{\,A_{\mathrm{std}}}\mathbf{a}_{A_0}(r))\!\downarrow_{\mathcal{U}'} \text{ enthält kei-}\\ \text{nen Teilterm der Form } f^{\,\langle\ldots,m,\ldots\rangle}(\ldots).\end{array}\right\}(*)$$

Hierbei bezeichnet $\mathcal{U}'$ die folgende A-Umgebung:

	Reg.
$\mathcal{U}'$:	$no\text{-}execute(\mathcal{R})$

D.9 „*Unfold right equality selector argument**“

78. Unfold right equality selector argument*

$$\frac{l=sel(f^{\,A}(t_1,t_2))}{\mathbf{env}_{\mathcal{U}'}(\mathbf{a}_{A_0}(l)=^{\,A_{\mathrm{std}}}sel^{\,A_{\mathrm{std}}}(\sigma_{\xi_{f(\ldots)}}(\mathbf{s}^*_f(\mathbf{un}^{u-1,0,m}_f(R_f)))))},$$

	Label	U-Lim.
A:	l	u

falls $u > 0$, $f \in (\Sigma \cap \Sigma_{\mathrm{C}}) \setminus \Sigma^{\mathrm{tail}}(P)$, $l \notin C$, $\sigma = \{x_1/\mathbf{a}_{A_0}(t_2), x_2/\mathbf{a}_{A_0}(t_1)\}$, $m = mark(\sigma)$ und folgendes gilt:

$$\left.\begin{array}{l}(\mathbf{a}_{A_0}(l)=^{\,A_{\mathrm{std}}}sel^{\,A_{\mathrm{std}}}(\sigma_{\xi_{f(\ldots)}}(\mathbf{s}^*_f(\mathbf{un}^{u-1,0,m}_f(R_f)))))\!\downarrow_{\mathcal{U}'} \text{ enthält kei-}\\ \text{nen Teilterm der Form } f^{\,\langle\ldots,m,\ldots\rangle}(\ldots).\end{array}\right\}(*)$$

Hierbei bezeichnet $\mathcal{U}'$ die folgende A-Umgebung:

	Reg.
$\mathcal{U}'$:	$no\text{-}execute(\mathcal{R})$

Anhang E

Zusätzliche Auswertungsregeln

In diesem Anhang definieren wir die Auswertungsregeln, die zwar in der Implementierung des Auswertungskalküls durch das ✓eriFun-System enthalten sind, deren Nützlichkeit aber durch unsere Fallstudien nicht mehr belegt wird.

E.1 „*Affirmative boolean functionality ind. hyp.*“

Affirmative boolean functionality induction hypothesis

$$\frac{f^{\,A}(l_1,\ldots,l_n)}{\texttt{if}(AND(\overline{\tilde{\sigma}_{\tilde{\xi}}(C)-lit}),\mathsf{env}_{\mathcal{U}'}(AND(d_1,\ldots,d_n)),\texttt{false})},\qquad \begin{array}{|c|}\hline \text{S-Lim.}\\ \hline A\text{:}\quad s\\ \hline\end{array}$$

falls $s>0$, $n>0$, $f^{\,A}(l_1,\ldots,l_n)\in \mathcal{ATerm}_{\texttt{bool}}(P)$, $f\in\Sigma(P)\setminus(\Sigma^{\text{case}}(P)\cup\Sigma_0^{=})$ und $p\neq\ominus$ gilt. Weiter muss ein $C\in\bigcup_{q\in\mathcal{Q}}\mathcal{Res}^*(\mathcal{C}_q)$ und ein $\tilde{\sigma}_{\tilde{\xi}}\in \mathit{inst\text{-}subst}_{\mathcal{U},H_{\mathrm{G}}}(C)$ existieren, so dass die folgenden Bedingungen erfüllt sind:

(1) $\exists lit\in\tilde{\sigma}_{\tilde{\xi}}(C)$ mit $lit=f(r_1,\ldots,r_n)$.

(2) C ist die Klausel einer Induktionshypothese.

(3) $\overline{\tilde{\sigma}_{\tilde{\xi}}(C)-lit}\subseteq H_{\mathrm{L}}\cup H_{\mathrm{G}}$.

(4) $s_{\text{new}}(1,|D|+1)\geq 1$ mit $D=\langle l_1{=}r_1,\ldots,l_n{=}r_n\rangle$.

(5) $\forall k\in\{1,\ldots,n\}$ gilt $\mathcal{U}'\vdash d_k\rightarrow\texttt{true}$ mit $\langle d_1,\ldots,d_n\rangle=\mathsf{pol}_{\oplus}(\mathsf{a}_{A'}(D))$.

Hierbei sind die A-Umgebung $\mathcal{U}'$ und die A-Annotation A' wie folgt gegeben:

Reg.
$\mathcal{U}'$: $\mathit{no\text{-}execute}(\mathcal{R})$

A_{top}	S-Lim.
A':	$s_{\text{new}}(1,\lvert D\rvert+1)$

E.2 „*Negative boolean functionality ind. hyp.*“

Negative boolean functionality induction hypothesis

$$\frac{f^{\,A}(l_1,\ldots,l_n)}{\neg(\mathtt{if}(AND(\overline{\tilde{\sigma}_{\tilde{\xi}}(C)-lit}),\mathsf{env}_{\mathcal{U}'}(\neg(AND(d_1,\ldots,d_n))),\mathtt{false}))},\qquad \begin{array}{|c|}\hline \text{S-Lim.}\\ \hline A\text{:}\quad s\\ \hline\end{array}$$

falls $s>0$, $n>0$, $f^{\,A}(l_1,\ldots,l_n)\in\mathcal{ATerm}_{\mathtt{bool}}(P)$, $f\in\Sigma(P)\setminus(\Sigma^{\mathrm{case}}(P)\cup\Sigma_0^{=})$ und $p\neq\oplus$ gilt. Weiter muss ein $C\in\bigcup_{q\in\mathcal{Q}}\mathcal{Res}^*(\mathcal{C}_q)$ und ein $\tilde{\sigma}_{\tilde{\xi}}\in \mathit{inst\text{-}subst}_{\mathcal{U},H_{\mathrm{G}}}(C)$ existieren, so dass die folgenden Bedingungen erfüllt sind:

(1) $\exists\neg lit\in\tilde{\sigma}_{\tilde{\xi}}(C)$ mit $lit=f(r_1,\ldots,r_n)$.

(2) C ist die Klausel einer Induktionshypothese.

(3) $\overline{\tilde{\sigma}_{\tilde{\xi}}(C)-lit}\subseteq H_{\mathrm{L}}\cup H_{\mathrm{G}}$.

(4) $s_{\mathrm{new}}(1,|D|+1)\geq 1$ mit $D=\langle l_1{=}r_1,\ldots,l_n{=}r_n\rangle$.

(5) $\forall k\in\{1,\ldots,n\}$ gilt $\mathcal{U}'\vdash d_k\to\mathtt{true}$ mit $\langle d_1,\ldots,d_n\rangle=\mathsf{pol}_{\oplus}(\mathsf{a}_{A'}(D))$.

Hierbei sind die A-Umgebung $\mathcal{U}'$ und die A-Annotation A' wie folgt gegeben:

Reg.
$\mathcal{U}'$: $\mathit{no\text{-}execute}(\mathcal{R})$

A_{top}	S-Lim.
A':	$s_{\mathrm{new}}(1,\lvert D\rvert+1)$

E.3 „*Left constructor functionality*“

Left constructor functionality

$$\frac{\mathit{cons}_i(t_1,\ldots,t_n){=}^{\,A}r}{\mathsf{env}_{\mathcal{U}'}(AND(d_1,\ldots,d_m))},\qquad \begin{array}{|c|}\hline \text{S-Lim.}\\ \hline A\text{:}\quad s\\ \hline\end{array}$$

falls $s>0$, $n>0$, $p\neq\ominus$ und die folgenden Bedingungen erfüllt sind:

(1) Für alle $j\in\{1,\ldots,n\}$ gilt $\mathit{sel}_{i,j}(\mathsf{e}(r))=t_j$ oder $\mathit{sel}_{i,j}(\mathsf{e}(r)){=}t_j\in_{\simeq_{\mathcal{U}}}H_{\mathrm{L}}\cup H_{\mathrm{G}}$.

(2) Es gilt $D=\{?\mathit{cons}_i(r)\}$ oder $D=\{\neg?\mathit{cons}_1(r),\ldots,\neg?\mathit{cons}_{i-1}(r),\neg?\mathit{cons}_{i+1}(r),\ldots,\neg?\mathit{cons}_n(r)\}$ sowie:

(a) $s_{\mathrm{new}}(1,|D|+1)\geq 1$.

(b) $\{d_1,\ldots,d_m\}=\mathsf{pol}_{\oplus}(\mathsf{a}_{A'}(D))$.

(c) $\forall k\in\{1,\ldots,m\}$ gilt $\mathcal{U}'\vdash d_k\to\mathtt{true}$.

Hierbei sind die A-Umgebung $\mathcal{U}'$ und die A-Annotation A' wie folgt gegeben:

Reg.
$\mathcal{U}'$: $\mathit{no\text{-}execute}(\mathcal{R})$

A_{top}	S-Lim.
A':	$s_{\mathrm{new}}(1,\lvert D\rvert+1)$

E.4 „*Right constructor functionality*“

Right constructor functionality

$$\frac{l =^{A} cons_i(t_1, \ldots, t_n)}{\mathbf{env}_{\mathcal{U}'}(AND(d_1, \ldots, d_m))}, \qquad \frac{\text{S-Lim.}}{A\text{:}\quad s}$$

falls $s > 0$, $n > 0$, $p \neq \ominus$ und die folgenden Bedingungen erfüllt sind:

(1) Für alle $j \in \{1, \ldots, n\}$ gilt $sel_{i,j}(\mathbf{e}(l)) = t_j$ oder $sel_{i,j}(\mathbf{e}(l)) =^{t_j} \in_{\simeq_{\mathcal{U}}} H_{\mathrm{L}} \cup H_{\mathrm{G}}$.

(2) Es gilt $D = \{?cons_i(l)\}$ oder $D = \{\neg ?cons_1(l), \ldots, \neg ?cons_{i-1}(l), \neg ?cons_{i+1}(l), \ldots, \neg ?cons_n(l)\}$ sowie:

   (a) $s_{\mathrm{new}}(1, |D| + 1) \geq 1$.

   (b) $\{d_1, \ldots, d_m\} = \mathbf{pol}_{\oplus}(\mathbf{a}_{A'}(D))$.

   (c) $\forall k \in \{1, \ldots, m\}$ gilt $\mathcal{U}' \vdash d_k \rightarrow \mathtt{true}$.

Hierbei sind die A-Umgebung $\mathcal{U}'$ und die A-Annotation A' wie folgt gegeben:

Reg.
$\mathcal{U}'$: $no\text{-}execute(\mathcal{R})$

A_{top}	S-Lim.
A':	$s_{\mathrm{new}}(1, \lvert D \rvert + 1)$

Anhang F

Zusätzliche Beispiele

In diesem Anhang sind die Beispiele zusammengefasst, die aus Gründen der Übersichtlichkeit nicht den Kapiteln 8-13 enthalten sind. Es sind insbesondere Beispiele für Überauswertungen und Endlos-Auswertungen.

F.1 $\mathtt{loop(2,y)}$

Ziel Das Beispiel zeigt, dass durch die Auswertung trivialer Prozeduraufrufe mit Hilfe der Regel „*Execute procedure call (constructor ground terms)*“ Endlos-Auswertungen entstehen können.

Beispiel Sei P das Programm $\langle D_{\mathtt{even}}, D_{\mathtt{ev}}, D_{\mathtt{loop}}\rangle$, wobei $D_{\mathtt{even}}$, $D_{\mathtt{ev}}$ und $D_{\mathtt{loop}}$ die Prozedurdefinitionen aus Abbildung F.1 bezeichnen. Seien weiter die Mengen der generalisierten Relationenbeschreibungen der Prozeduren wie folgt gegeben:

$$\begin{aligned} \mathit{grds}(\mathtt{even}) &= \{\{\langle\langle \neg\mathtt{?0(x)}, \neg\mathtt{?0(pred(x))}\rangle, \{\{\mathtt{x/pred(pred(x))}\}\}\rangle\}\}, \\ \mathit{grds}(\mathtt{ev}) &= \{\{\langle\langle \neg\mathtt{?0(y)}\rangle, \{\{\mathtt{y/pred(y)}\}\}\rangle\}\}, \end{aligned}$$

$$\begin{aligned} \mathit{grds}(\mathtt{loop}) = \{\{&\langle\langle \neg\mathtt{?0(x)}, \neg\mathtt{?0(pred(x))}, \mathtt{ev(x,y)}\rangle, \{\{\mathtt{x/pred(pred(x))}\}\}\rangle, \\ &\langle\langle \neg\mathtt{?0(x)}, \neg\mathtt{?0(pred(x))}, \neg\mathtt{ev(x,y)}\rangle, \{\{\mathtt{x/succ(x))}\}\}\rangle\}\}. \end{aligned}$$

Der Prozeduraufruf $\mathtt{loop(2,y)}$ ist offensichtlich ein trivialer Prozeduraufruf und wir können ihn daher mit Hilfe der Regel „*Execute procedure call (constructor ground terms)*“ auswerten. Es ergibt sich die folgende Endlos-Auswertung:

$$\begin{aligned} \mathcal{U} \vdash\ & \underline{\mathtt{loop(2,y)}} && \rightarrow \\ & \mathtt{if}(\underline{\mathtt{?0(2)}}, \mathtt{0}, \mathtt{if}(\mathtt{?0(pred(2))}, \mathtt{0}, \mathtt{if}(\mathtt{ev(2,y)}, \ldots, \mathtt{loop(3,y)}))) && \rightarrow \\ & \underline{\mathtt{if}(\mathtt{false}, \mathtt{0}, \mathtt{if}(\mathtt{?0(pred(2))}, \mathtt{0}, \mathtt{if}(\mathtt{ev(2,y)}, \ldots, \mathtt{loop(3,y)})))} && \rightarrow \\ & \mathtt{if}(\mathtt{?0}(\underline{\mathtt{pred(2)}}), \mathtt{0}, \mathtt{if}(\mathtt{ev(2,y)}, \mathtt{loop}(\mathtt{pred}(\mathtt{pred(2)}), \mathtt{y}), \mathtt{loop(3,y)})) && \rightarrow \\ & \mathtt{if}(\underline{\mathtt{?0(1)}}, \mathtt{0}, \mathtt{if}(\mathtt{ev(2,y)}, \mathtt{loop}(\mathtt{pred}(\mathtt{pred(2)}), \mathtt{y}), \mathtt{loop(3,y)})) && \rightarrow \\ & \underline{\mathtt{if}(\mathtt{false}, \mathtt{0}, \mathtt{if}(\mathtt{ev(2,y)}, \mathtt{loop}(\mathtt{pred}(\mathtt{pred(2)}), \mathtt{y}), \mathtt{loop(3,y)}))} && \rightarrow \\ & \mathtt{if}(\mathtt{ev(2,y)}, \mathtt{loop}(\mathtt{pred}(\underline{\mathtt{pred(2)}}), \mathtt{y}), \mathtt{loop(3,y)}) && \rightarrow \\ & \mathtt{if}(\mathtt{ev(2,y)}, \mathtt{loop}(\underline{\mathtt{pred(1)}}, \mathtt{y}), \mathtt{loop(3,y)}) && \rightarrow \end{aligned}$$

$$
\begin{array}{ll}
\mathtt{if}(\mathtt{ev}(2,\mathtt{y}),\underline{\mathtt{loop}(0,\mathtt{y})},\mathtt{loop}(3,\mathtt{y})) & \rightarrow \\
\mathtt{if}(\mathtt{ev}(2,\mathtt{y}),\mathtt{if}(\underline{\mathtt{?0}(0)},0,\ldots),\mathtt{loop}(3,\mathtt{y})) & \rightarrow \\
\mathtt{if}(\mathtt{ev}(2,\mathtt{y}),\underline{\mathtt{if}(\mathtt{true},0,\ldots)},\mathtt{loop}(3,\mathtt{y})) & \rightarrow \\
\mathtt{if}(\mathtt{ev}(2,\mathtt{y}),0,\underline{\mathtt{loop}(3,\mathtt{y})}) & \rightarrow \\
\quad\vdots & \vdots \\
\mathtt{if}(\mathtt{ev}(2,\mathtt{y}),0,\mathtt{if}(\mathtt{ev}(3,\mathtt{y}),0,\underline{\mathtt{loop}(4,\mathtt{y})})) & \rightarrow \\
\quad\vdots & \vdots
\end{array}
$$

Der Grund für die Endlos-Auswertung ist, dass wir während der symbolischen Auswertung des Prozeduraufrufs $\mathtt{loop}(2,\mathtt{y})$ den eigentlich irrelevanten rekursiven Aufruf $\mathtt{loop}(3,\mathtt{y})$ betrachten, da die Bedingung $\mathtt{ev}(2,\mathtt{y})$ nicht zu $\mathtt{true}$ ausgewertet werden kann.

F.2 $\mathtt{quot}'(1,\mathtt{y})$

Ziel Das Beispiel zeigt, dass triviale Prozeduraufrufe durch die Regel „*Execute procedure call (constructor ground terms)*“ nicht notwendigerweise rekursionsvollständig ausgewertet werden können.

Beispiel Sei P das Programm $\langle D_{\mathtt{minus}'}, D_{\mathtt{quot}'}\rangle$, wobei $D_{\mathtt{minus}'}$ und $D_{\mathtt{quot}'}$ die Prozedurdefinitionen aus Abbildung F.2 bezeichnen. Der Prozeduraufruf $\mathtt{quot}'(1,\mathtt{y})$ ist dann offensichtlich trivial und es ergibt sich folgende symbolische Auswertung:

$$
\begin{array}{lll}
\mathcal{U} & \vdash & \underline{\mathtt{quot}'(1,\mathtt{y})} \rightarrow \\
 & & \mathtt{if}(\mathtt{?0}(\mathtt{y}),1,\mathtt{if}(\underline{\mathtt{y>1}},0,\mathtt{if}(\mathtt{?0}(1),0,\mathtt{succ}(\mathtt{quot}'(\mathtt{minus}'(1,\mathtt{y}),\mathtt{y})))))) \rightarrow \\
 & & \quad\vdots \\
 & & \mathtt{if}(\mathtt{?0}(\mathtt{y}),1,\mathtt{if}(\mathtt{?0}(\mathtt{pred}(\mathtt{y})),\mathtt{succ}(\mathtt{quot}(\mathtt{minus}(1,\mathtt{y}),\mathtt{y})),0)).
\end{array}
$$

Der Term $\mathtt{if}(\mathtt{?0}(\mathtt{y}),1,\mathtt{if}(\mathtt{?0}(\mathtt{pred}(\mathtt{y})),\mathtt{succ}(\mathtt{quot}(\mathtt{minus}(1,\mathtt{y}),\mathtt{y})),0))$ kann nicht weiter vereinfacht werden. Der Grund für das Scheitern der Auswertung der rekursiven Aufrufe des Prozeduraufrufs $\mathtt{quot}'(1,\mathtt{y})$ ist, dass wir das Argument $\mathtt{minus}'(1,\mathtt{y})$ der Induktionsvariablen im rekursiven Aufruf $\mathtt{quot}'(\mathtt{minus}'(1,\mathtt{y}),\mathtt{y})$ nicht zu einem Konstruktorgrundterm vereinfachen können. Die Auswertung des Prozeduraufrufs $\mathtt{quot}'(1,\mathtt{y})$ ist daher nicht sinnvoll.

F.3 $\mathtt{quot}(1,\mathtt{y})$

Ziel Das Beispiel zeigt, dass eine Beschränkung der Auswertung trivialer Prozeduraufrufe $f(t_1,\ldots,t_n)$ durch die Regel *Execute procedure call (constructor ground terms)* auf Prozeduraufrufe, die in den Argumenten der relevanten Variablen einer Relationenbeschreibung $R \in \mathit{grds}(f)$ Konstruktorgrundterme enthalten, sinnvolle Auswertungen verhindert.

Beispiel Sei P das Programm $\langle D_{\mathtt{minus}}, D_{\mathtt{quot}}\rangle$, wobei $D_{\mathtt{minus}}$ und $D_{\mathtt{quot}}$ die entsprechenden Prozedurdefinitionen aus Abbildung 9.1 bezeichnen. Seien weiter die

$$
\begin{array}{rcl}
D_{\mathtt{even}} & = & \mathtt{function\ even(x : nat) : bool} \Leftarrow \\
& & \quad \mathtt{if(?0(x),} \\
& & \qquad \mathtt{true,} \\
& & \qquad \mathtt{if(?0(pred(x)),} \\
& & \qquad\quad \mathtt{false,} \\
& & \qquad\quad \mathtt{even(pred(pred(x)))))}
\end{array}
$$

$$
\begin{array}{rcl}
D_{\mathtt{ev}} & = & \mathtt{function\ ev(x : nat, y : nat) : nat} \Leftarrow \\
& & \quad \mathtt{if(?0(y),} \\
& & \qquad \mathtt{even(x),} \\
& & \qquad \mathtt{ev(x, pred(y)))}
\end{array}
$$

$$
\begin{array}{rcl}
D_{\mathtt{loop}} & = & \mathtt{function\ loop(x : nat, y : nat) : nat} \Leftarrow \\
& & \quad \mathtt{if(?0(x),} \\
& & \qquad \mathtt{0,} \\
& & \qquad \mathtt{if(?0(pred(x)),} \\
& & \qquad\quad \mathtt{0,} \\
& & \qquad\quad \mathtt{if(ev(x, y),} \\
& & \qquad\qquad \mathtt{loop(pred(pred(x)), y),} \\
& & \qquad\qquad \mathtt{loop(succ(x), y)))).}
\end{array}
$$

Abbildung F.1: Prozedurdefinitionen für `even`, `ev` und `loop`.

Mengen der generalisierten Relationenbeschreibungen der Prozeduren wie folgt gegeben:

$$
\begin{array}{rcl}
\mathit{grds}(\mathtt{minus}) & = & \{\{\langle\langle \neg \mathtt{?0(x)} \rangle, \{\{\mathtt{x/pred(x)}\}\}\rangle\}, \\
& & \quad \{\langle\langle \neg \mathtt{?0(y)} \rangle, \{\{\mathtt{y/pred(y)}\}\}\rangle\}\}
\end{array}
$$

$$
\mathit{grds}(\mathtt{quot}) = \{\{\langle\{\neg \mathtt{?0(y)}, \neg(\mathtt{y>x})\}, \{\{\mathtt{x/minus(x,y)}\}\}\rangle\}.
$$

Den trivialen Prozeduraufruf `quot(1, y)` können wir mit Hilfe der Regel „*Execute procedure call (constr. ground terms)*“ wie folgt symbolisch auswerten:

$$
\begin{array}{lll}
\mathcal{U} \vdash & \underline{\mathtt{quot(1,y))}} & \rightarrow \\
& \mathtt{if(?0(y), 1, if(}\underline{\mathtt{y>1}}\mathtt{, 0, if(?0(1), 0, succ(quot(minus(1, y), y)))))} & \rightarrow \\
& \qquad\qquad \vdots & \vdots \\
& \mathtt{if(?0(y), 1, if(?0(pred(y)), succ(quot(}\underline{\mathtt{minus(1,y)}}\mathtt{, y)), 0))} & \rightarrow \\
& \qquad\qquad \vdots & \vdots \\
& \mathtt{if(?0(y), 1, if(?0(pred(y)), succ(quot(}\underline{\mathtt{minus(0, pred(y))}}\mathtt{, y)), 0))} & \rightarrow \\
& \qquad\qquad \vdots & \vdots \\
& \mathtt{if(?0(y), 1, if(?0(pred(y)), succ(}\underline{\mathtt{quot(0,y)}}\mathtt{), 0))} & \rightarrow \\
& \qquad\qquad \vdots & \vdots \\
& \mathtt{if(?0((y), 1, if(?0((pred(y)), 1, 0)).} &
\end{array}
$$

Würden wir die Auswertung trivialer Prozeduraufrufe wie oben beschrieben einschränken, so wäre die Auswertung von `quot(1, y)` nicht möglich, da sowohl `x` als auch `y` relevante Variablen der Relationenbeschreibung von `quot` sind. Wir könnten also die Prozedur `quot` nicht entfernen.

$$
\begin{array}{rcl}
D_{\texttt{minus'}} & = & \texttt{function minus'(x : nat, y : nat) : nat} \Leftarrow \\
& & \quad \texttt{if(?0(y),} \\
& & \qquad \texttt{x,} \\
& & \qquad \texttt{if(?0(x),} \\
& & \qquad\quad \texttt{0,} \\
& & \qquad\quad \texttt{pred(minus'(x, pred(y)))))}
\end{array}
$$

$$
\begin{array}{rcl}
D_{\texttt{quot'}} & = & \texttt{function quot'(x : nat, y : nat) : nat} \Leftarrow \\
& & \quad \texttt{if(?0(y),} \\
& & \qquad \texttt{x,} \\
& & \qquad \texttt{if(y>x,} \\
& & \qquad\quad \texttt{0,} \\
& & \qquad\quad \texttt{if?0(x),} \\
& & \qquad\qquad \texttt{0,} \\
& & \qquad\qquad \texttt{quot'(minus'(x, y), y))))}
\end{array}
$$

Abbildung F.2: Prozedurdefinitionen für `minus'` und `quot'`.

F.4 butlast(l)

Ziel Das Beispiel zeigt, dass durch die Verwendung der lokalen Hypothesen während der Auswertung der Recursion-Guards in der Regel „*Execute procedure call (recursive cases)*“ Endlos-Auswertungen auftreten können. Wir nehmen daher für das Beispiel an, dass zur Auswertung der Recursion-Guards auch lokale Hypothesen verwendet werden dürfen.

Beispiel Sei P das Programm $\langle D_{\texttt{list}}, D_{\texttt{butlast}} \rangle$, wobei $D_{\texttt{list}}$ die entsprechende Typoperatordefinition aus Abbildung 3.1 bezeichnet und $D_{\texttt{butlast}}$ die entsprechende Prozedurdefinition aus Abbildung 12.3. Sei weiter die Menge der generalisierten Relationenbeschreibungen von `butlast` gegeben durch $\{R\}$, wobei gilt

$$R = \{\langle\langle \neg\texttt{?empty(l)}\rangle, \{\{\texttt{l/tl(l)}\}\}\rangle\}\,.$$

Wir wollen nun das folgende Lemma beweisen:

$$
\begin{array}{l}
\texttt{lemma butlast\_not\_equal} \Leftarrow \texttt{all l : list[@a]} \\
\quad \texttt{if(?empty(l), true, } \neg\texttt{(butlast(l)=l))}\,.
\end{array}
$$

Zum Beweis dieses Lemmas ist es notwendig, die Gültigkeit der Sequenz

$$\emptyset, \emptyset \vdash \texttt{if(?empty(l), true, } \neg\texttt{(butlast(l)=l))}$$

nachzuweisen. Der Teilterm `butlast(l)` schlägt hierzu eine Induktion über die Relationenbeschreibung R vor. Für diese Induktion erhalten wir als Induktionsschritt die folgende Sequenz:

$$
\begin{array}{c}
\{\neg\texttt{?empty(l)}\}, \{\texttt{if(?empty(tl(l)), true, } \neg\texttt{(butlast(tl(l))=tl(l)))}\} \vdash \\
\texttt{if(?empty(l), true, } \neg\texttt{(butlast(l)=l))}\,.
\end{array}
$$

Zum Beweis dieser Sequenz werten wir den Goal-Term bzgl. der A-Umgebung $\mathcal{U}$ aus, wobei wir die Hypothese der Sequenz als globale Hypothese verwenden. Es ergibt sich dann die folgende symbolische Auswertung:

$\mathcal{U} \vdash$ if(?empty(l), true, ¬(butlast(l)=l)) $\xrightarrow{(1)}$

if(false, true, ¬(butlast(l)=l)) $\xrightarrow{(2)}$

¬(butlast(l)=l) $\xrightarrow{(3)}$

¬(if(?empty(l),
empty,
if(?empty(tl(l)),
empty,
add(hd(l), butlast(tl(l)))))=l) $\xrightarrow{(4)}$

¬(if(false, ..., ...)) $\xrightarrow{(5)}$

¬(if(?empty(tl(l)), empty, add(hd(l), butlast(tl(l))))=l) $\xrightarrow{(6)}$

if(?empty(tl(l)),
empty,
add(hd(l), if(?empty(tl(l)),
empty,
if(?empty(tl(tl(l))),
empty,
add(hd(tl(l)), butlast(tl(tl(l)))))))=l) $\xrightarrow{(7)}$

if(?empty(tl(l)),
empty,
add(hd(l), if(true,
empty,
if(?empty(tl(tl(l))),
empty,
add(hd(tl(l)), butlast(tl(tl(l)))))))=l) $\xrightarrow{(8)}$

if(?empty(tl(l)),
empty,
add(hd(l), if(?empty(tl(tl(l))),
empty,
add(hd(tl(l)), butlast(tl(tl(l))))))=l) $\xrightarrow{(9)}$

⋮ ⋮

In Schritt (6) kann aufgrund der lokalen Hypothese ¬?empty(tl(l)) der Prozeduraufruf butlast(tl(l)) ausgewertet werden. Diese Auswertung führt die neue if-Bedingung ?empty(tl(tl(l))) in den Term ein. Mit Hilfe dieser if-Bedingung können wir den rekursiven Prozeduraufruf

butlast(tl(tl(l)))

in Schritt (9) auswerten. Diese Auswertung führt wieder eine if-Bedingung ein, die wieder die Auswertung eines rekursiven Prozeduraufrufs ermöglicht. Dies setzt sich endlos fort.

F.5 foo(x)

Ziel Das Beispiel zeigt, dass die Auswertung einer Prozedur f in einem der Recursion-Guards der Prozedur während der Regel „*Execute procedure call (recursive cases)*“ zu Endlos-Auswertungen führen kann.

Beispiel Sei P das Programm $\langle D_{\texttt{foo}} \rangle$, wobei $D_{\texttt{foo}}$ die folgende Prozedurdefinition bezeichnet:

$$
\begin{array}{rcl}
D_{\texttt{foo}} & = & \texttt{function foo(x : nat) : nat} \Leftarrow \\
 & & \quad \texttt{if(?0(x),} \\
 & & \quad\quad \texttt{0,} \\
 & & \quad\quad \texttt{if(foo(pred(x))>pred(x),} \\
 & & \quad\quad\quad \texttt{42,} \\
 & & \quad\quad\quad \texttt{foo(foo(pred(x)))))).}
\end{array}
$$

Weiter sei die Menge *grds*(`foo`) der generalisierten Relationenbeschreibungen von `foo` gegeben durch $\{R\}$, wobei gilt

$$
\begin{array}{l}
R = \{\langle\langle \neg\texttt{?0(x)}, \neg(\texttt{foo(pred(x))>pred(x)})\rangle, \{\{\texttt{x/pred(x)}\}, \{\texttt{x/foo(pred(x))}\}\}\rangle, \\
\quad \langle\langle \neg\texttt{?0(x)}, \texttt{foo(pred(x))>pred(x)}\rangle, \{\{\texttt{x/pred(x)}\}\}\rangle\}\,.
\end{array}
$$

Unter der Annahme, dass die Prozedur `foo` im Recursion-Guard von `foo`, während der Überprüfung der Anwendungsbedingungen der Regel „*Execute procedure call (recursive cases)*“ ausgewertet werden darf, führt die symbolische Auswertung von `foo(x)` zu einem endlosen rekursiven Abstieg: Um zu Überprüfen, ob der Term `foo(x)` mit Hilfe der Regel „*Execute procedure call (recursive cases)*“ ausgewertet werden kann, muss der instantiierte Recursion-Guard

$$AND(\neg\texttt{?0(x)}, \neg(\texttt{foo(pred(x))>pred(x)}))$$

symbolisch ausgewertet werden. Während dieser Auswertung wird überprüft, ob der Prozeduraufruf `foo(pred(x))` ausgewertet werden kann. Dies führt erneut zu einer symbolischen Auswertung des Recursion-Guards. Dieser rekursive Abstieg setzt sich endlos fort.

Anmerkung Die Bedingung `foo(pred(x))>pred(x)` der Prozedurdefinition von `foo` im Beispiel ist für die Terminierung der Prozedur irrelevant und die Relationenbeschreibung R somit nicht optimal. Die Bedingung ist außerdem auch für die Semantik der Prozedur irrelevant. Es scheint daher so, dass unser Beispiel ausgesprochen künstlich ist und Prozeduren wie `foo` mit entsprechenden nicht-optimalen generalisierten Relationenbeschreibungen in echten Fallstudien nicht auftreten. Dies ist jedoch aufgrund der von ✓`eriFun` verwendeten Terminierungsanalyse nicht der Fall. Die Terminierungsanalyse des Systems basiert auf einer Erweiterung des „*Estimation Calculus*“ für unvollständig definierte Prozeduren [48, 57]. Sie kann lediglich starke Terminierung von Prozeduren nachweisen. Das bedeutet, dass die Terminierung der Prozedur unabhängig von der Semantik der Prozedur sein muss [49]. Deshalb werden bei nicht-stark terminierenden Prozeduren zusätzliche redundante Bedingungen in die Prozedurrümpfe eingefügt (im Beispiel `foo(pred(x))>pred(x)`) mittels derer die starke Terminierung der Prozeduren sichergestellt wird. Ein Beispiel für eine solche Prozedur in einer echten Fallstudie ist die Prozedur `eval` aus [55].

F.6 `ordered(minsort(k))`

Ziel Das Beispiel zeigt, dass die Auswertung eines Prozeduraufrufs durch die Regel „*Execute procedure call (recursive cases)*“ zu Endlos-Auswertungen führen kann, falls die Argumente der Induktionsvariablen in den rekursiven Aufrufen ausgewertet werden. Wir nehmen daher für dieses Beispiel an, dass bei der Auswertung eines Prozeduraufrufs die Labels der Argumente in den rekursiven Aufrufen auf $\perp$ gesetzt werden.

$\mathcal{U} \vdash$ ordered(minsort(k))

$\vdots$

ordered(add(minimum(k), minsort(delete(minimum(k), k)))) $\overset{(1)}{\uparrow}$

$\vdots$

ordered(add(..., minsort(delete(minimum(k), k)))) $\overset{(2)}{\uparrow}$

$\vdots$

ordered(add(..., minsort(delete(if(?empty(tl(k)), hd(k), if(minimum(tl(k))>hd(k), hd(k), minimum(tl(k)))), k)))) $\overset{(3)}{\uparrow}$

$\vdots$

ordered(add(..., minsort(if(?empty(tl(k)), delete(hd(k), k), delete(if(minimum(tl(k))>hd(k), hd(k), minimum(tl(k))), k))))) $\overset{(4)}{\uparrow}$

$\vdots$

ordered(add(..., minsort(if(?empty(tl(k)), tl(k), delete(if(minimum(tl(k))>hd(k), hd(k), minimum(tl(k))), k))))) $\overset{(5)}{\uparrow}$

$\vdots$

ordered(add(..., minsort(if(?empty(tl(k)), tl(k), if(minimum(tl(k))>hd(k), delete(hd(k), k), delete(minimum(tl(k)), k)))))) $\overset{(6)}{\uparrow}$

$\vdots$

ordered(add(..., minsort(if(?empty(tl(k)), tl(k), if(minimum(tl(k))>hd(k), tl(k), delete(minimum(tl(k)), k)))))) $\overset{(7)}{\uparrow}$

$\vdots$

ordered(add(..., minsort(if(?empty(tl(k)), tl(k), if(minimum(tl(k))>hd(k), tl(k), delete(minimum(tl(k)), k)))))) $\overset{(8)}{\uparrow}$

$\vdots$

ordered(add(..., minsort(if(?empty(tl(k)), tl(k), if(...,tl(k), if(minimum(tl(k))=hd(k), tl(k), add(hd(k), delete(minimum(tl(k)), tl(k)))))))))) $\overset{(9)}{\uparrow}$

$\vdots$

ordered(add(..., if(?empty(tl(k)), M, if(..., M, minsort(add(hd(k), delete(minimum(tl(k)), tl(k)))))))) $\overset{(10)}{\uparrow}$

$\vdots$

Abbildung F.3: Symbolische Auswertung des Goal-Terms der Sequenz (F.1) aus Beispiel F.6. Hierbei bezeichnet M den Term minsort(tl(k)).

Beispiel Sei P das Programm $\langle D_{\texttt{list}}, D_{\texttt{delete}}, D_{\texttt{minimum}}, D_{\texttt{ordered}}, D_{\texttt{minsort}}\rangle$, wobei $D_{\texttt{list}}$ die Typoperatordefinition aus Abbildung 3.1 bezeichnet und $D_{\texttt{delete}}$, $D_{\texttt{minimum}}$, $D_{\texttt{ordered}}$ und $D_{\texttt{minsort}}$ die Prozedurdefinitionen aus Abbildung 9.3. Seien weiter die Mengen der generalisierten Relationenbeschreibungen der Prozeduren gegeben durch $grds(\texttt{delete}) = grds(\texttt{minimum}) = grds(\texttt{ordered}) = \{R_1\}$ und $grds(\texttt{minsort}) = \{R_2\}$, wobei gilt

$$\begin{aligned} R_1 &= \{\langle\langle \neg\texttt{?empty(l)}\rangle, \{\{\texttt{l/tl(l)}\}\}\rangle\}, \\ R_2 &= \{\langle\langle \neg\texttt{?empty(l)}\rangle, \{\{\texttt{l/delete(minimum(l),l)}\}\}\rangle\}. \end{aligned}$$

Wir wollen nun das folgende Lemma beweisen:

$$\begin{array}{l}\texttt{lemma minsort\_sorts} \Leftarrow \texttt{all l : nat} \\ \quad \texttt{ordered(minsort(l))}.\end{array}$$

Zum Beweis dieses Lemmas ist es notwendig, die Gültigkeit der Sequenz

$$\emptyset, \emptyset \vdash \texttt{ordered(minsort(l))}$$

nachzuweisen. Der Teilterm `minsort(l)` schlägt hierzu eine Induktion über die Relationenbeschreibung R_2 vor. Für diese Induktion erhalten wir als Induktionsschritt die folgende Sequenz:

$$\begin{array}{c}\{\neg\texttt{?empty(l)}\}, \{\texttt{ordered(minsort(delete(minimum(l),l)))}\} \vdash \\ \texttt{ordered(minsort(l))}\,.\end{array} \tag{F.1}$$

Zum Beweis der Sequenz werten wir den Goal-Term bzgl. der A-Umgebung $\mathcal{U}$ aus, wobei wir wie üblich die Hypothese der Sequenz als globale Hypothese verwenden. Es ergibt sich dann die in Abbildung F.3 dargestellte Endlos-Auswertung. Betrachten wir die Endlos-Auswertung ein wenig genauer. Durch Auswertung des Arguments `delete(minimum(tl(k)),k)` des rekursiven Aufrufs von `minsort` in Auswertungsschritt (8) wird unter anderem der Term

$$\texttt{add(hd(k),delete(minimum(tl(k)),tl(k)))}$$

eingeführt. Dieser Term führt schließlich in Auswertungsschritt (10) dazu, dass wir `minsort` erneut auswerten. Diese Auswertung führt wieder zu einer Auswertung der Argumente des rekursiven Aufrufs und somit zu einer Auswertung des rekursiven Aufrufs selbst. Dies setzt sich endlos fort.

F.7 ?0(goo(x, y))

Ziel Das Beispiel zeigt, dass es notwendig ist den Ergebnisterm der „Unfold"-Regeln mit der A-Umgebung $\mathcal{U}'$ zu annotieren, um sicherzustellen, dass bei einer erfolgreichen Überprüfung der Anwendungsbedingung $(*)$ auf jeden Fall die rekursiven Prozeduraufrufe von f eliminiert werden können. Wir gehen daher in diesem Beispiel davon aus, dass die Ergebnisterme der „*Unfold*"-Regeln nicht mit der A-Umgebung $\mathcal{U}'$ annotiert werden.

Beispiel Sei P das Programm $\langle D_{\texttt{foo}}, D_{\texttt{goo}}\rangle$, wobei $D_{\texttt{foo}}$ und $D_{\texttt{goo}}$ die folgenden Prozedurdefinitionen bezeichnen:

$$
\begin{array}{rcl}
D_{\texttt{foo}} & = & \texttt{function foo(x : nat) : nat} \Leftarrow \\
 & & \quad \texttt{if(?0(x),} \\
 & & \quad\quad \texttt{1} \\
 & & \quad\quad \texttt{foo(pred(x)))}
\end{array}
$$

$$
\begin{array}{rcl}
D_{\texttt{goo}} & = & \texttt{function goo(x : nat, y : nat) : nat} \Leftarrow \\
 & & \quad \texttt{if(?0(x),} \\
 & & \quad\quad \texttt{0} \\
 & & \quad\quad \texttt{if(?0(y),} \\
 & & \quad\quad\quad \texttt{foo(goo(pred(x), foo(y))),} \\
 & & \quad\quad\quad \texttt{0))}\,.
\end{array}
$$

Die jeweiligen Mengen der generalisierten Relationenbeschreibungen und die jeweiligen Execution-Guards der Prozeduren seien wie folgt gegeben:

$$
\begin{array}{rcl}
\mathit{grds}(\texttt{foo}) & = & \{\langle\langle \neg\texttt{?0(x)}\rangle, \{\{\texttt{x/pred(x)}\}\}\}, \\
\mathit{grds}(\texttt{goo}) & = & \{\langle\langle \neg\texttt{?0(x)}\rangle, \{\{\texttt{x/pred(x)}\}\}\},
\end{array}
$$

$$
\begin{array}{rcl}
\mathit{exec}_{\texttt{foo}} & = & \texttt{?0(x)}, \\
\mathit{exec}_{\texttt{goo}} & = & \texttt{if(?0(x), true, } \neg\texttt{?0(y)).}
\end{array}
$$

Wir notieren im Folgenden für die Terme die relevanten Unfold-Limits. Alle weiteren Komponenten der A-Annotationen werden nicht dargestellt. Betrachten wir nun den folgenden Term:

$$
\texttt{if(?0(foo}^{\,2}\texttt{(goo}^{\,2}\texttt{(pred(x), foo}^{\,2}\texttt{(y)))), ?0(goo}^{\,2}\texttt{(x, y)), true)} \tag{F.2}
$$

Für den Prozeduraufruf $\texttt{?0(goo}^{\,2}\texttt{(x, y))}$ können wir die Regel „*Unfold structure test argument*“ anwenden, da der instantiierte Prozedurrumpf ohne Verwendung der „*Execute*“-Regeln wie folgt ausgewertet werden kann:[1]

$$
\begin{array}{lll}
\mathcal{U}' \vdash & \underline{\texttt{?0(if(?0(x), 0, if(?0(y), foo}^{\,1}\texttt{(goo}^{\,0}\texttt{(pred(x), foo}^{\,1}\texttt{(y))), 0)))}} & \rightarrow \\
 & \texttt{if(?0(x),} \underline{\texttt{?0(0)}}\texttt{, ?0(if(?0(y), foo}^{\,1}\texttt{(goo}^{\,0}\texttt{(pred(x), foo}^{\,1}\texttt{(y))), 0)))} & \rightarrow \\
 & \texttt{if(?0(x), true,} \underline{\texttt{?0(if(?0(y), foo}^{\,1}\texttt{(goo}^{\,0}\texttt{(pred(x), foo}^{\,1}\texttt{(y))), 0)))}} & \rightarrow \\
 & \texttt{if(?0(x), true, if(?0(y),} \underline{\texttt{?0(foo}^{\,1}\texttt{(goo}^{\,0}\texttt{(pred(x), foo}^{\,1}\texttt{(y))))}}\texttt{, ?0(0))))} & \rightarrow \\
 & \texttt{if(?0(x), true, if(?0(y), true,} \underline{\texttt{?0(0)}}\texttt{)))} & \rightarrow \\
 & \texttt{if(?0(x), true,} \underline{\texttt{if(?0(y), true, true)}}\texttt{)} & \rightarrow \\
 & \underline{\texttt{if(?0(x), true, true)}} & \rightarrow \\
 & \texttt{true.}
\end{array}
$$

Für den Term (F.2) ergibt sich dann insgesamt die in Abbildung F.4 dargestellte symbolische Auswertung. Es ist zu beachten, dass bei dieser Auswertung die „*Execute*“-Regeln verwendet werden durften. Das Ergebnis der symbolischen Auswertung enthält offensichtlich immer noch den rekursiven Aufruf der Prozedur `goo`. Das Problem ist, dass der Prozeduraufruf `foo(y)` im rekursiven Aufruf von `goo` mit Hilfe

[1] Die lokale Hypothesenmenge von $\mathcal{U}'$ enthält hierbei die Hypothese $\texttt{?0(foo(goo(pred(x),}\ \mathit{foo}(y))))$.

der „*Execute*“-Regeln während der eigentlichen symbolischen Auswertung in Auswertungsschritt (2) ausgewertet werden kann, wohingegen eine solche Auswertung während der Überprüfung der Anwendungsbedingung ($*$) nicht möglich ist. Durch diese Auswertung von `foo(y)` geht jedoch die Übereinstimmung mit der lokalen Hypothese `?0(foo(goo(pred(x),foo(y))))` verloren. Wir können daher den rekursiven Aufruf nicht eliminieren.

F.8 `if(?0(x),false,x>plus(x,y))`

Ziel Das Beispiel zeigt, dass die generelle Verwendung der Regel „*Execute procedure call (recursive – no additional case analysis)*“ im Zusammenspiel mit der Regel „*Execute procedure call (recursive cases)*“ zu Endlos-Auswertungen führen kann. Wir nehmen daher für dieses Beispiel an, dass wir die Anwendung der Regel „*Execute procedure call (recursive – no additional case analysis)*“ generell erlauben.

Beispiel Sei P das Programm $P = \langle D_{\texttt{plus}} \rangle$, wobei $D_{\texttt{plus}}$ die entsprechende Prozedurdefinition aus Abbildung 8.2 bezeichnet. Es ergibt sich dann die folgende Endlos-Auswertung:[2]

$$
\begin{array}{lll}
\mathcal{U} \vdash & \texttt{if(?0(x),false,x>}\underline{\texttt{plus(x,y)}}\texttt{)} & \overset{(\square)}{\rightarrow} \\
& \vdots & \vdots \\
& \texttt{if(?0(x),false,}\underline{\texttt{x>succ(plus(pred(x),y))}}\texttt{)} & \overset{(\triangle)}{\rightarrow} \\
& \vdots & \vdots \\
& \texttt{if(?0(x),false,if(?0(pred(x)),false,pred(x)>}\underline{\texttt{plus(pred(x),y)}}\texttt{))))} & \overset{(\square)}{\rightarrow} \\
& \vdots & \vdots \\
& \texttt{if(?0(x),} & \\
& \quad \texttt{false,} & \overset{(\triangle)}{\rightarrow} \\
& \quad \texttt{if(?0(pred(x)),false,}\underline{\texttt{pred(x)>succ(plus(pred(pred(x)),y))}}\texttt{)))))} & \\
& \vdots & \vdots
\end{array}
$$

[2] Die Auswertungen von Prozeduraufrufen aufgrund der Regel „*Execute procedure call (recursive – no additional case analysis)*“ haben wir hierbei mit ($\square$) markiert und die Auswertungen aufgrund der Regel „*Execute procedure call (recursive cases)*“ mit ($\triangle$).

$$
\begin{aligned}
\mathcal{U} \vdash\ & \texttt{if}(\texttt{?0}(\texttt{foo}^{\,2}(\texttt{goo}^{\,2}(\texttt{pred}(\texttt{x}), \texttt{foo}^{\,2}(\texttt{y})))), \underline{\texttt{?0}(\texttt{goo}^{\,2}(\texttt{x}, \texttt{y}))}, \texttt{true}) && \overset{(1)}{\rightarrow} \\
& \texttt{if}(\texttt{?0}(\texttt{foo}^{\,2}(\texttt{goo}^{\,2}(\texttt{pred}(\texttt{x}), \texttt{foo}^{\,2}(\texttt{y})))), \texttt{?0}(\texttt{if}(\texttt{?0}(\texttt{x}), \texttt{0}, \texttt{if}(\texttt{?0}(\texttt{y}), \texttt{foo}^{\,1}(\texttt{goo}^{\,0}(\texttt{pred}(\texttt{x}), \underline{\texttt{foo}^{\,1}(\texttt{y})}), \texttt{0}))), \texttt{true}) && \overset{(2)}{\rightarrow} \\
& \texttt{if}(\texttt{?0}(\texttt{foo}^{\,2}(\texttt{goo}^{\,2}(\texttt{pred}(\texttt{x}), \texttt{foo}^{\,2}(\texttt{y})))), \texttt{?0}(\texttt{if}(\texttt{?0}(\texttt{x}), \texttt{0}, \texttt{if}(\texttt{?0}(\texttt{y}), \texttt{foo}^{\,1}(\texttt{goo}^{\,0}(\texttt{pred}(\texttt{x}), \texttt{if}(\underline{\texttt{?0}(\texttt{y})}, \texttt{0}, \texttt{succ}(\texttt{foo}^{\,1}(\texttt{pred}(\texttt{y}))))), \texttt{0}))), \texttt{true}) && \overset{(3)}{\rightarrow} \\
& \texttt{if}(\texttt{?0}(\texttt{foo}^{\,2}(\texttt{goo}^{\,2}(\texttt{pred}(\texttt{x}), \texttt{foo}^{\,2}(\texttt{y})))), \texttt{?0}(\texttt{if}(\texttt{?0}(\texttt{x}), \texttt{0}, \texttt{if}(\texttt{?0}(\texttt{y}), \texttt{foo}^{\,1}(\texttt{goo}^{\,0}(\texttt{pred}(\texttt{x}), \underline{\texttt{if}(\texttt{true}, \texttt{0}, \texttt{succ}(\texttt{foo}^{\,1}(\texttt{pred}(\texttt{y})))))}, \texttt{0}))), \texttt{true}) && \overset{(4)}{\rightarrow} \\
& \texttt{if}(\texttt{?0}(\texttt{foo}^{\,2}(\texttt{goo}^{\,2}(\texttt{pred}(\texttt{x}), \texttt{foo}^{\,2}(\texttt{y})))), \underline{\texttt{?0}(\texttt{if}(\texttt{?0}(\texttt{x}), \texttt{0}, \texttt{if}(\texttt{?0}(\texttt{y}), \texttt{foo}^{\,1}(\texttt{goo}^{\,0}(\texttt{pred}(\texttt{x}), \texttt{0})), \texttt{0})))}, \texttt{true}) && \overset{(5)}{\rightarrow} \\
& \texttt{if}(\texttt{?0}(\texttt{foo}^{\,2}(\texttt{goo}^{\,2}(\texttt{pred}(\texttt{x}), \texttt{foo}^{\,2}(\texttt{y})))), \texttt{if}(\texttt{?0}(\texttt{x}), \underline{\texttt{?0}(\texttt{0})}, \texttt{?0}(\texttt{if}(\texttt{?0}(\texttt{y}), \texttt{foo}^{\,1}(\texttt{goo}^{\,0}(\texttt{pred}(\texttt{x}), \texttt{0})), \texttt{0}))), \texttt{true}) && \overset{(6)}{\rightarrow} \\
& \texttt{if}(\texttt{?0}(\texttt{foo}^{\,2}(\texttt{goo}^{\,2}(\texttt{pred}(\texttt{x}), \texttt{foo}^{\,2}(\texttt{y})))), \texttt{if}(\texttt{?0}(\texttt{x}), \texttt{true}, \underline{\texttt{?0}(\texttt{if}(\texttt{?0}(\texttt{y}), \texttt{foo}^{\,1}(\texttt{goo}^{\,0}(\texttt{pred}(\texttt{x}), \texttt{0})), \texttt{0}))}), \texttt{true}) && \overset{(7)}{\rightarrow} \\
& \texttt{if}(\texttt{?0}(\texttt{foo}^{\,2}(\texttt{goo}^{\,2}(\texttt{pred}(\texttt{x}), \texttt{foo}^{\,2}(\texttt{y})))), \texttt{if}(\texttt{?0}(\texttt{x}), \texttt{true}, \texttt{if}(\texttt{?0}(\texttt{y}), \texttt{?0}(\texttt{foo}^{\,1}(\texttt{goo}^{\,0}(\texttt{pred}(\texttt{x}), \texttt{0})))), \underline{\texttt{?0}(\texttt{0})})), \texttt{true}) && \overset{(8)}{\rightarrow}^{!} \\
& \texttt{if}(\texttt{?0}(\texttt{foo}^{\,2}(\texttt{goo}^{\,2}(\texttt{pred}(\texttt{x}), \texttt{foo}^{\,2}(\texttt{y})))), \texttt{if}(\texttt{?0}(\texttt{x}), \texttt{true}, \texttt{if}(\texttt{?0}(\texttt{y}), \texttt{?0}(\texttt{foo}^{\,1}(\texttt{goo}^{\,0}(\texttt{pred}(\texttt{x}), \texttt{0})))), \texttt{?0}(\underline{\texttt{0}}))), \texttt{true}) \\
& \texttt{if}(\texttt{?0}(\texttt{foo}^{\,2}(\texttt{goo}^{\,2}(\texttt{pred}(\texttt{x}), \texttt{foo}^{\,2}(\texttt{y})))), \texttt{if}(\texttt{?0}(\texttt{x}), \texttt{true}, \texttt{if}(\texttt{?0}(\texttt{y}), \texttt{?0}(\texttt{foo}^{\,1}(\texttt{goo}^{\,0}(\texttt{pred}(\texttt{x}), \texttt{0})))), \texttt{true})), \texttt{true}).
\end{aligned}
$$

Abbildung F.4: Symbolische Auswertung des Terms (F.2) aus Beispiel F.7.

Literaturverzeichnis

[1] M. Aderhold. Formula Generalization in ✓veriFun. Diplomarbeit, Fachgebiet Programmiermethodik, Technische Universität Darmstadt, 2004.

[2] P.B. Andrews. *An Introduction to Mathematical Logic and Type Theory: To Truth through Proof.* Computer Science and Applied Mathematics. Academic Press, 1986.

[3] K. Arnold, J. Gosling, and D. Holmes. *The Java Programming Language.* Addison-Wesley Professional. Addison-Wesley, Reading, MA, 4th edition, 2000.

[4] R. Aubin. *Mechanizing Structural Induction.* Ph.D. Thesis, University of Edingburgh, 1976.

[5] J. Auer. Erweiterung des ✓veriFun-Systems um Funktionen höherer Ordnung, Diplomarbeit. Diplomarbeit, Fachgebiet Programmiermethodik, Technische Universität Darmstadt, 2006.

[6] J. Avenhaus. *Reduktionssysteme.* Springer, Heidelberg, 1995.

[7] F. Baader and T. Nipkow. *Term Rewriting and All That.* Cambridge University Press, Cambridge, 1998.

[8] D. Basin and T. Walsh. Difference Matching. In D. Kapur, editor, *Automated Deduction - CADE-11*, volume 607 of *Lecture Notes in Computer Science*, pages 295–309. Springer, 1992.

[9] S. Biundo, B. Hummel, D. Hutter, and C. Walther. The Karlsruhe Induction Theorem Proving System. In *Proceedings of the 8th CADE*, volume 230 of *Lecture Notes in Computer Science*. Springer, 1986.

[10] R. S. Boyer and J S. Moore. *A Computational Logic.* Academic Press, New York, 1979.

[11] A. Bundy. The Automation of Proof by Mathematical Induction. In A. Robinson and A. Voronkov, editors, *Handbook of Automated Reasoning*, volume 1, pages 845–911. Elsevier Science, 2001.

[12] A. Bundy, A. Stevens, , F. v. Harmelen, A. Ireland, and A. Smaill. The Oyster-CLAM System. In *Proceedings of the 10th CADE*, Lecture Notes in Artificial Intelligence. Springer, 1990.

[13] A. Bundy, A. Stevens, F. van Harmelen, A. Ireland, and A. Smaill. Rippling: A Heuristic for Guiding Inductive Proofs. *Artifical Intelligence*, 62(2):185–253, 1993.

[14] C.-L. Chang and R. C.-T Lee. *Symbolic Logic and Mechanical Theorem Proving.* Computer Science Classics. Academic Press, 1987.

[15] N. Dershowitz and D.A. Plaisted. Rewriting. In A. Robinson and A. Voronkov, editors, *Handbook of Automated Reasoning*, volume 1, chapter 9, pages 535–610. Elsevier Science, 2001.

[16] H.-D. Ebbinghaus. *Einführung in die Mengenlehre.* B.I. Wissenschaftsverlag, Mannheim, 3rd edition, 1994.

[17] H.-D. Ebbinghaus, J. Flum, and W. Thomas. *Einführung in die mathematische Logik.* Spektrum Akademischer Verlag, Heidelberg, 4th edition, 1996.

[18] D. Endros. Erweiterung des Symbolischen Auswertens von ✓eriFun um Injektivität und Unifikation. Studienarbeit, Fachgebiet Programmiermethodik, Technische Universität Darmstadt, 2005.

[19] E. Gamma and Beck K. *Contributing to Eclipse. Principles, Patterns and Plug-Ins.* The Eclipse Series. Addison-Wesley, Reading, MA, 2004.

[20] M. Gonder. Entwurf und Implementierung kontextabhängiger Prozeduren und Untertypen in ✓eriFun. Diplomarbeit, Fachgebiet Programmiermethodik, Technische Universität Darmstadt, 2006.

[21] M. J. C. Gordon and T. F. Melham. *Introduction to HOL: A Theorem Proving Environment for Higher Order Logic.* Cambridge University Press, Cambridge, 1993.

[22] P. Graf. *Term Indexing*, volume 1053 of *Lecture Notes in Artificial Intelligence.* Springer, Heidelberg, 1996.

[23] D. Gries and F.B. Schneider. Avoiding the Undefined by Underspecification. In J. van Leeuwen, editor, *Computer Science Today: Recent Trends and Developments*, volume 1000 of *Lecture Notes in Computer Science*, pages 366–373. Springer, 1995.

[24] P. R. Halmos. *Naive Set Theory.* Springer, Heidelberg, 1998.

[25] D. Hutter. *Mustergesteuerte Strategien für das Beweisen von Gleichungen.* Dissertation, Universität Karlsruhe, 1991.

[26] D. Hutter. Annotated Reasoning. *Annals of Mathematics and Artifical Intelligence (AMAI). Special Issue on Strategies in Automated Deduction*, 29, 2000.

[27] M. Kaufmann and J S. Moore. ACL2: An Industrial Strength Theorem Prover for a Logic Based on Common Lisp. *IEEE Transaction on Software Engineering*, 23(4):203–213, 1997.

[28] Michael Kohlhase. OMDOC: Towards an Internet Standard for the Administration, Distribution, and Teaching of Mathematical Knowledge. In *AISC*, pages 32–52, 2000.

[29] A. Krauss. Programmverifikation mit Modulen und Axiomatischen Spezifikationen in ✓eriFun. Diplomarbeit, Fachgebiet Programmiermethodik, Technische Universität Darmstadt, 2005.

[30] J. Loeckx, H.-D. Ehrich, and M. Wolf. *Specification of Abstract Data Types.* Wiley-Teubner, New York, 1996.

[31] D. W. Loveland. *Automated Theorem Proving: A Logical Basis.* North Holland, Amsterdam, 1978.

[32] W. McCune. Experiments with Discrimination-Tree Indexing and Path Indexing for Term Retrieval. *Journal of Automated Reasoning*, 9(2):147–167, 1992.

[33] N. Müller. OMDoc-Repräsentation von Programmen und Beweisen in ✓ `eriFun`. Diplomarbeit, Fachgebiet Programmiermethodik, Technische Universität Darmstadt, 2005.

[34] S. Owre, J. M. Rushby, and N. Shankar. PVS: A Prototype Verification System. In Deepak Kapur, editor, *Automated Deduction - CADE-11*, volume 607 of *Lecture Notes in Computer Science*, pages 748–752. Springer, 1992.

[35] L. C. Paulson. The Foundation of a Generic Theorem Prover. *Journal of Automated Reasoning*, 5(3):363–397, 1989.

[36] I.V. Ramakrishnan, R. Sekar, and A. Voronkov. Term indexing. In A. Robinson and A. Voronkov, editors, *Handbook of Automated Reasoning*, volume 2, chapter 26, pages 1853–1964. Elsevier Science, 2001.

[37] R. Ramesh, I. Ramakrishnan, and D. Warren. Automata-Driven Indexing of Prolog Clauses. *Journal of Logic Programming*, 23(2):151–202, 1995.

[38] W. Reif. The KIV-Approach to Software Verification. In M. Broy and S. Jähnichen, editors, *KORSO - Methods, Languages and Tools for the Construction of Correct Software*, volume 1009 of *Lecture Notes in Computer Science*, pages 339–370. Springer, 1995.

[39] A. Schlosser, C. Walther, M. Gonder, and M. Aderhold. Context dependent procedures and computed types in ✓`eriFun`. In *Workshop Programming Languages meets Program Verification (PLPV-06), Proceedings*, pages 72–86, 2006.

[40] U. Schöning. *Logik für Informatiker*. Spektrum Akademischer Verlag, Heidelberg, 5th edition, 2000.

[41] U. Schöning. *Theoretische Informatik - kurzgefasst*. Spektrum Akademischer Verlag, Heidelberg, 4th edition, 2001.

[42] M. Schubart. ✓`eriFun` - Entwurf und Implementierung eines interaktiven Induktionsbeweisers zur Programmverifikation. Diplomarbeit, Fachgebiet Programmiermethodik, Technische Universität Darmstadt, 1999.

[43] M. Stickel. The Path-Indexing Method for Indexing Terms. Technical Note 473, Artificial Intelligence Center, SRI International, Menlo Park, CA, 1983.

[44] D. Szallies. Ein Werkzeug zur automatischen Widerlegung von Aussagen in ✓`eriFun`. Diplomarbeit, Fachgebiet Programmiermethodik, Technische Universität Darmstadt, 2006.

[45] K. Walrath, M. Campione, A. Huml, and S. Zakhour. *The JFC Swing Tutorial: A Guide to Constructing GUIs*. Addison-Wesley Professional. Addison-Wesley, Reading, MA, 2nd edition, 2004.

[46] C. Walther. Computing Induction Axioms. In *Logic Programming and Automated Reasoning, International Conference LPAR 1992*, volume 624 of *Lecture Notes in Computer Science*, pages 381–392. Springer, 1992.

[47] C. Walther. Mathematical Induction. In D.M. Gabbay, C.J. Hogger, and J.A. Robinson, editors, *Handbook of Logic in Artificial Intelligence and Logic Programming*, volume 2, pages 127–227. Oxford University Press, 1994.

[48] C. Walther. On Proving the Termination of Algorithms by Machine. *Artifical Intelligence*, 71(1):101–157, 1994.

[49] C. Walther. Criteria for Termination. In S. Hölldobler, editor, *Intellectics and Computational Logic*, pages 361–386. Kluwer Academic Publishers, Dordrecht, 2000.

[50] C. Walther. Recursion, Induction, Verification. Vorlesungsskriptum, Fachgebiet Programmiermethodik, Technische Universität Darmstadt, 2004.

[51] C. Walther and S. Schweitzer. The ✓`eriFun` Tutorial. Bericht VFR 02/04, Fachgebiet Programmiermethodik, Technische Universität Darmstadt, 2002.

[52] C. Walther and S. Schweitzer. ✓`eriFun` User Guide. Bericht VFR 02/01, Fachgebiet Programmiermethodik, Technische Universität Darmstadt, 2002.

[53] C. Walther and S. Schweitzer. A Machine Supported Proof of the Unique Prime Factorization Theorem. Bericht VFR 02/03, Programmiermethodik, Technische Universität Darmstadt, 2002.

[54] C. Walther and S. Schweitzer. A Verification of Binary Search. Bericht VFR 02/02, Programmiermethodik, Technische Universität Darmstadt, 2002.

[55] C. Walther and S. Schweitzer. A Machine-Verified Code Generator. In *Logic for Programming, Artificial Intelligence, and Reasoning, 10th International Conference, LPAR 2003, Almaty, Kazakhstan, September 22-26, 2003, Proceedings*, volume 2850 of *Lecture Notes in Computer Science*, pages 91–106. Springer, 2003.

[56] C. Walther and S. Schweitzer. About ✓`eriFun`. In *Automated Deduction - CADE-19*, volume 2741 of *Lecture Notes in Computer Science*, pages 322–327. Springer, 2003.

[57] C. Walther and S. Schweitzer. Automated Termination Analysis for Incompletely Defined Programs. In *Logic for Programming, Artificial Intelligence and Reasoning (LPAR-11)*, volume 3452 of *Lecture Notes in Computer Science*, pages 332–346. Springer, 2004.

[58] C. Walther and S. Schweitzer. Verification in the Classroom. *Journal of Automated Reasoning*, 32:35–73, 2004.

[59] C. Walther and S. Schweitzer. Reasoning about Incompletely Defined Programs. In *Logic for Programming, Artificial Intelligence and Reasoning (LPAR-12)*, volume 3835 of *Lecture Notes in Artificial Intelligence*, pages 427–442. Springer, 2005.

Definitionsverzeichnis

Regelverzeichnis - Namen

Regelverzeichnis - Nummern

Symbolverzeichnis

Sachverzeichnis

Lebenslauf

Persönliche Daten

Name	Dirk Stephan Schweitzer
Geburtsdatum	3. Mai 1974
Geburtsort	Frankfurt am Main

Schulausbildung

1980 – 1984	Grundschule Rödelheim am Biedenkopfer Weg in Frankfurt am Main
1984 – 1990	Helene-Lange-Schule (Gymnasium) in Frankfurt am Main
1990 – 1993	Friedrich-Dessauer-Gymnasium Frankfurt am Main

Akademischer Werdegang

1993 – 1996	Studium der Informatik mit Nebenfach Mathematik an der Universität Karlsruhe (TH) Abschluss des Vordiploms
1996 – 2000	Studium der Informatik mit Nebenfach Mathematik an der TU Darmstadt
1999	Abschluss als Diplom-Informatiker
1999 - 2005	Wissenschaftlicher Mitarbeiter am FG Programmiermethodik der TU Darmstadt.